华恒科技

HHARM2440 开发平台技术手册



目 录

版 本 声 明

本手册适用于 HHARM2440 系列平台，最近更新的手册一般情况下适用于以前的同一种型号 CPU 的其它平台。

由于安装我们提供的光盘上的开发环境后，会在您的装有 Redhat Linux 的 PC 机的根目录下生成 HHARM2440-Rx(x 为 1,2,3…)这样的目录和其它目录，以下说明与目录有关的操作以 HHARM2440 目录进行说明，实际的目录以安装光盘后建立的目录名为准。

因 HHARM2440 平台有很多款，本手册中对硬件平台及 PCB 图的描述与实际套件可能有差别，主要包括相关硬件、硬件平台的电路图及软件源代码部分，以您收到的套件及我们提供的光盘中的内容为准，如有异议，可以立即和我们取得联系（联系方式，请见本手册最后一页)。

本手册的说明主要是针对华恒提供的 HHARM2440 系列产品，手册中难免存在一些错误及不足，敬请客户谅解，并真诚地欢迎您提出宝贵的意见和建议，我们定期作出修改。

修 订 描 述

日期	修订版本	描述	作者
2005 年 2 月 2 日	1.1	初稿，参考了 HHARM2410 平台的手册写法	王康
2005 年 2 月 28 日	1.2	移植 minigui 最新的移植文件及采用新的模板,加入了 IDE 的操作	王康
2005 年 7 月 19 日	1.3	因从原来的 2.4.18 内核升级到 2.4.20，有的接口驱动需要重新移植	王康
2005 年 8 月 9 日	1.4	内核由 2.4.18 升级到 2.4.20	王康
2005 年 11 月 8 日	1.5	建立/HHARM2440/modules.TestApp 目录,加入 jffs2 和 cramfs 文件系统,测试外围接口规范了描述,公司地址更改	王康

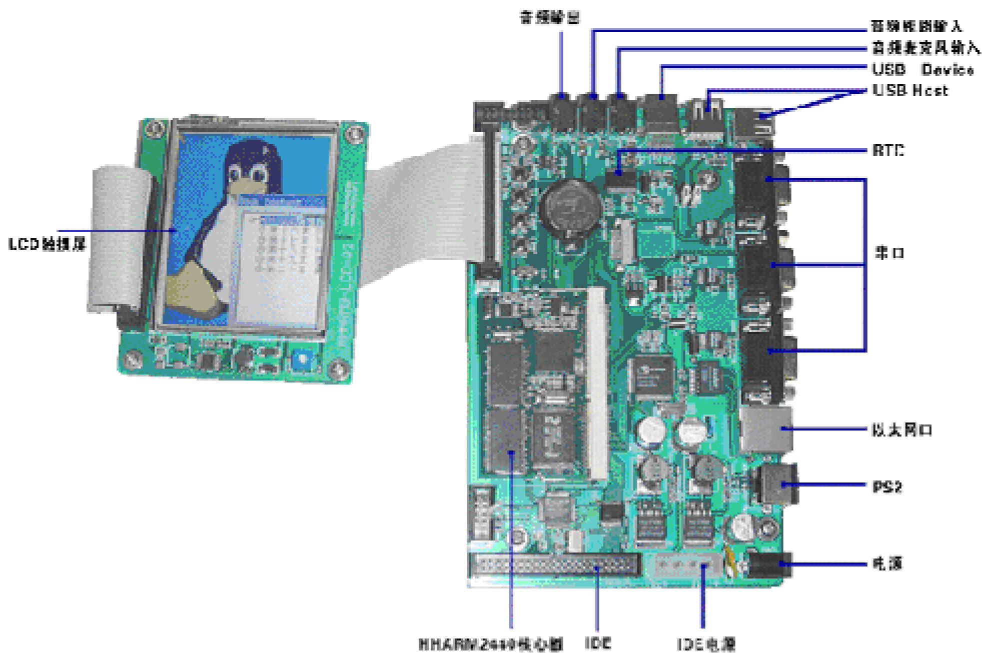
第一章 快速上手指南

HHARM2440 套件的标准配置：		
1	核心板	一块
2	底板（外设接口板）	一块
3	240x320 16 位色 LCD 模块(LCD、LCD 底板)	一块
4	JTAG 烧写器（PCB 板和数据线）	一块
5	串口线	一根
6	软件安装光盘	一张
7	9V (**220V**,50Hz,1000mA) /12V(**100-220V,50/60Hz,3A**)电源适配器	一台
8	配置清单	一张

感谢您选择华恒科技的产品，在本章中您可以对 HHARM2440 开发板有个初步的认识，对开发板进行简单检测及建立开发环境。

开发板的简单测试

HHARM2440 开发板连接图：LCD 上显示的图形是 miniGUI 的一个演示示例。

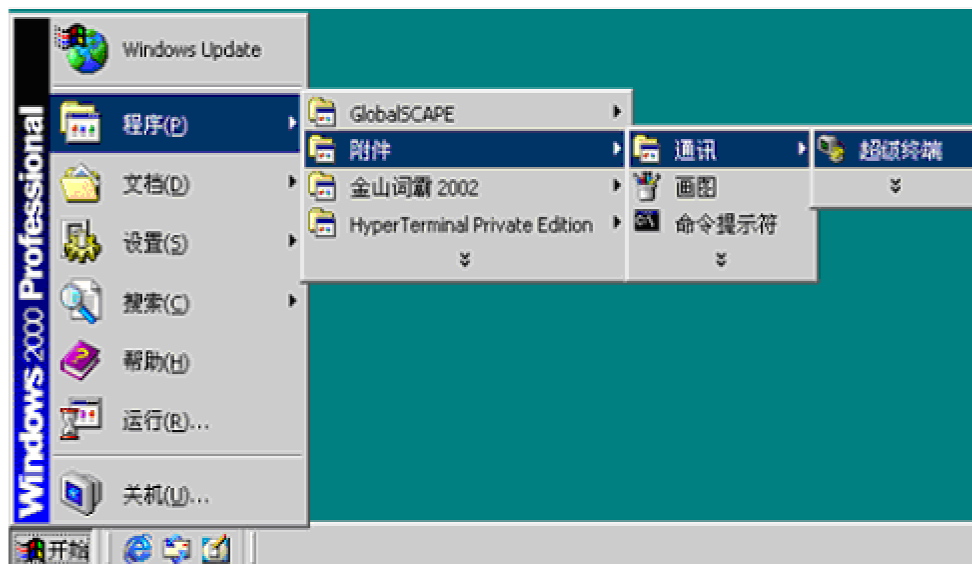


在本章节中您可以对 HHARM2440 开发套件有个初步的认识，对开发板进行

简单检测及建立开发环境。

Windows 98 和 Windows 2000 安装盘中都附有超级终端应用程序。如果您的系统没有安装超级终端，您可以在控制面板“添加/删除程序”一栏选择安装超级终端应用程序。

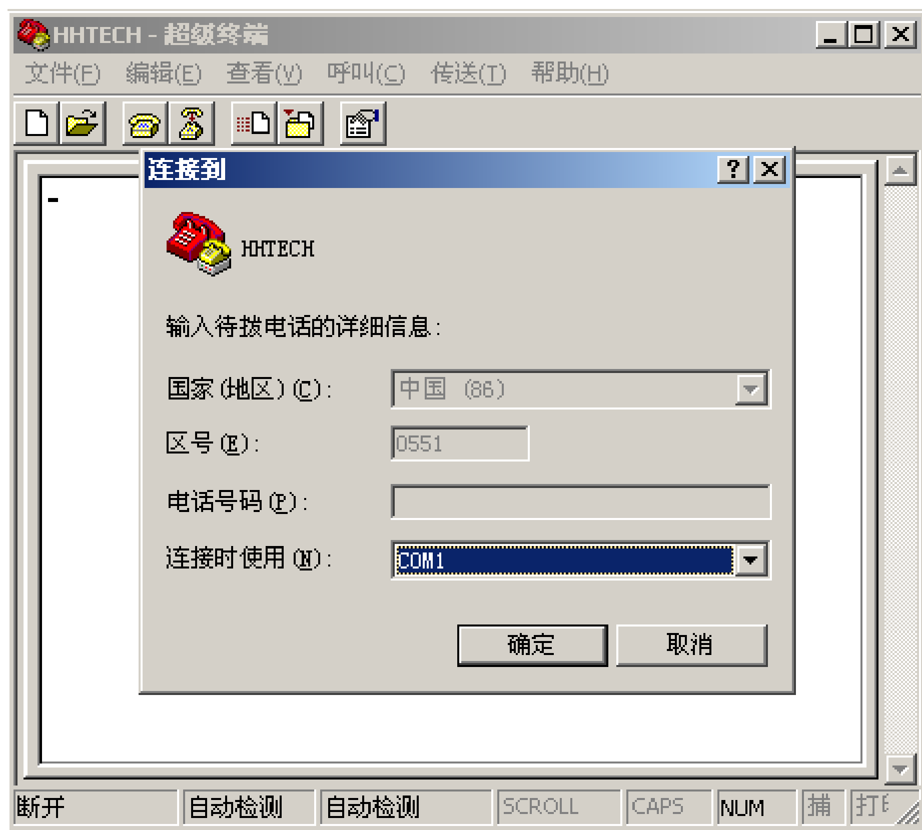
首先我们在 Windows 2000 环境下启动超级终端。



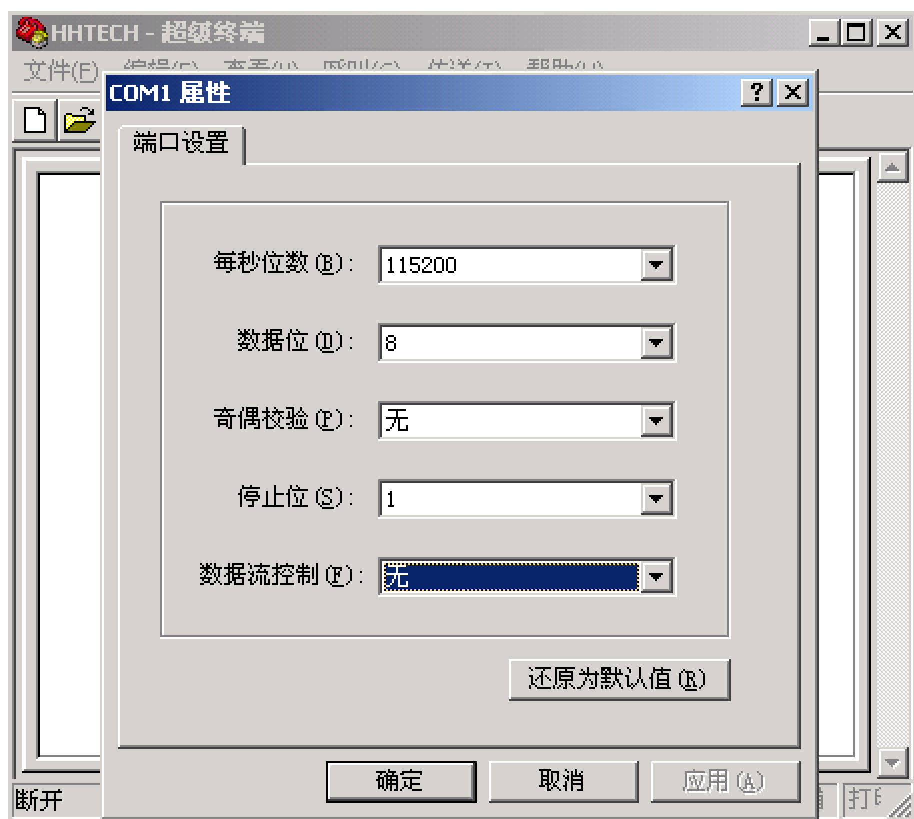
进入超级终端。



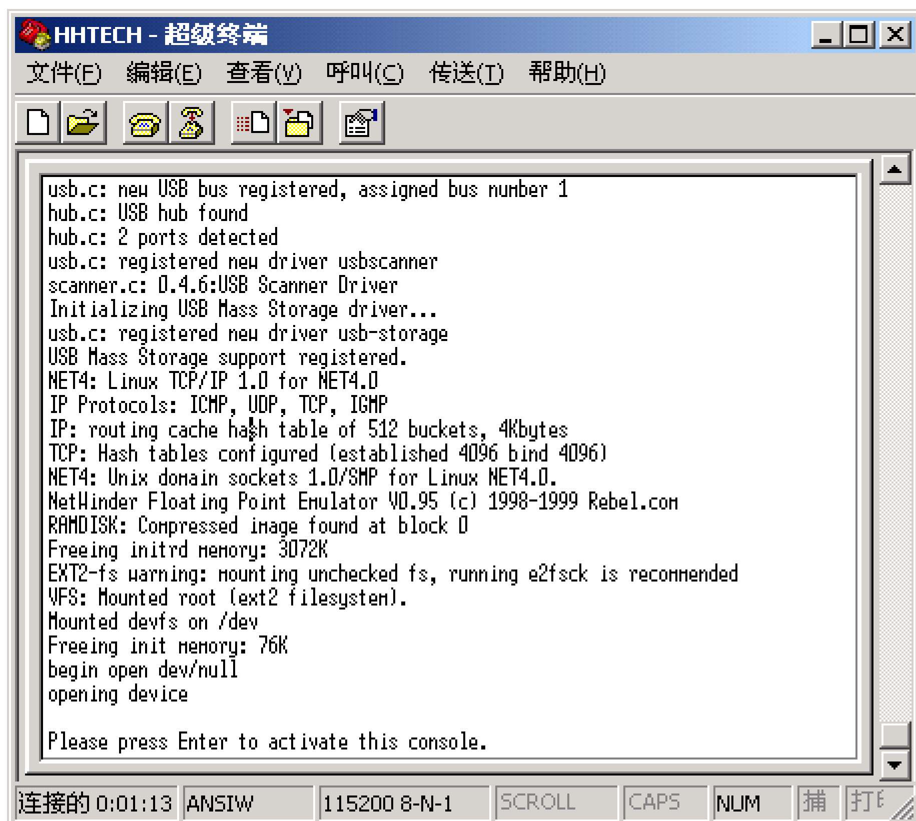
输入连接名称，这个名称可以随便输入，您可以输入“ HHTECH ”，然后点击“ 确定 ”按钮。设置连接使用串口 1（COM1），点击“ 确定 ”进入 COM1 的属性设置窗口。



设置串口每秒位数为 115200,数据位 8,奇偶校验无,停止位 1,数据流控"无"。确定后进入超级终端窗口。



请您用产品附带的串口对接线将开发板与 PC 机的串口 1 连接。接通开发板 9V/12V 电源后超级终端中将会显示开发板的启动信息：



并最终进入 Linux 的 shell 命令提示符（有时需要按一下 Ctrl + C，再键入 Enter(回车键)才能见到提示)。现在就可以键入 linux 的命令操作开发板了，例如可以键入 ls 命令列出开发板已经存在哪些目录了。

HHTECH - 超级终端

文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

```
~ # ls
bin      dev      font     lib      mnt      proc     tmp      var
cramfs   etc      jffs2    linuxrc  modules  sbin     usr
~ # ifconfig
ifconfig: Warning: cannot open /proc/net/dev. Limited output.: No such file or d
irectory
eth0      Link encap:Ethernet  HWaddr 00:13:F6:6C:87:89
          inet addr:192.168.2.121  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Base address:0x300

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1

~ # ping 192.168.2.97
PING 192.168.2.97 (192.168.2.97): 56 data bytes
64 bytes from 192.168.2.97: icmp_seq=0 ttl=64 time=0.5 ms
64 bytes from 192.168.2.97: icmp_seq=1 ttl=64 time=0.3 ms

--- 192.168.2.97 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.4/0.5 ms
~ #
```

连接的 0:06:51 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印

如果您对 Linux 操作系统比较熟悉，那就直接在终端提示符下键入 minicom，启动 minicom 这个终端程序，设置串口波特率为 115200，同样可以看到启动信息和操作开发板，minicom 的详细配置介绍请见本手册的附录。

若按以上操作步骤启动开发板时工作不正常，请立即拨打电话与华恒公司联系(联系方式请见本手册的最后一页)，我们将为您提供指导意见或更换开发板。您也可以到我们网站上去看一看 FAQ(常见问题解答)：

www.hhcn.com/chinese/hharmfaq.html ：华恒 HHARM 系列嵌入式 LINUX 开发套件常见问题解答。

【注意】禁止带电拔插串口对接线和 JTAG 烧写器！

第二章 产 品 简 介

HHARM2440 是华恒科技推出的用于高端手持设备、微型智能控制设备的开发套件。采用400MHz(以后采用533MHz) 的ARM920T 内核的处理器S3C2440。该款套件核心板的尺寸仅相当于名片的2/3 大小，尺寸如此小巧的嵌入式核心板是国内首创。

S3C2440 内部集成了微处理器和一些手持设备的常用外围组件，特别适用于手持产品。同时S3C2440微处理器也是一款多用途的通用芯片，其内部集成的常用外围组件，可用于各种领域。为应用于手持设备的低成本实现，提供了更高性价比。

HHARM2440 套件由核心板和底板（外设板或称基本板）组成，核心板上集成Samsung S3C2440 处理器，64M SDRAM以及16M 的FLASH，为您的应用研发提供了足够的空间。底板上则提供以下外设接口：两个四线RS-232 串口(COM1)，一个两线串口，两个USB接口，一个10M/100M自适应以太网接口，一个TFT LCD接口，一个触摸屏接口。核心板和底板配合即构成一个最小的完整应用系统。系统具有体积小、耗电低、处理能力强、等特点，能够装载和运行嵌入式Linux 操作系统。用户可以在这个系统平台上进行自主软件开发。HHARM2440 套件中提供底板硬件电路图及硬件设计文档，极大的方便了用户进行硬件扩展开发。

HHARM2440套件提供完备的嵌入式Linux 开发环境及丰富的开发调试工具软件。

S3C2440 微处理器的精彩特性：

- ARM920T 嵌入式处理器内核，主频可达533MHz；
- 扩展总线最大频率 136MHz；
- 32 位数据,27 位外部地址线 ；
- 完全静态设计(0-203M) ；
- 存储控制器(八个存储体) ：
  包含 RAM(SDRAM)控制器,NAND 控制器 ；
  复位时引导芯片选择(8-，16-比特存储或 NAND 可供选择) ；
- 四个带有 PWM 的 16 位定时器
- 多达 60 个中断源的中断控制器 ；
- RTC；
- 三个 UART ，Supports IrDA 1.0；
- 四个 DMA 通道；(支持外设 DMA)
- 8通道，500KSPS，10-bit ADC；
- 支持 STN . 与 TFT LCD 控制器；

- 看门狗；
- **AC97**音频编码器和解码器接口；
- 两个USB HOST接口，一个USB Device接口；
- IIC-Bus接口；
- 两个串行外围接口电路（SPI）
- SD卡接口；

HHARM2440 开发套件硬件主要结构：

Sumsung S3C2440 处理器
16Mbytes 16 位 FLASH
64Mbytes 32 位 SDRAM
两个四线 RS-232 串口，一个两线 RS-232 串口
一个 10M/100M 自适应以太网接口
一个 TFT LCD 接口,
一个触摸屏接口
一个 IDE 硬盘接口
音频接口
JTAG 接口
9V 直流电源（需要客户自己购买）
H/W 复位建
运行状态指示 LED 灯
更详细的接口请见底板的电路图。

第三章 软 件 系 统

HHARM2440 为一台采用 S3C2440 处理器、提供 RS232 接口，安装有 Linux 操作系统的软硬件开发平台，其功用相当于一台装有 Redhat Linux，装有串口的 PC 机。对于 HHARM2440,它提供的所有软件（操作系统和应用软件）都固化在板上 FLASH 里面，就相当于 PC 机的硬盘。FLASH 上的内容，包括 bootloader 都可通过烧写工具来更新升级。用户可为开发应用程序或更改其上的操作系统工作方式（因为操作系统是开放源代码的），和在 PC 上开发应用唯一的不同之处在于它要采用一种交叉编译的开发模式，即为 HHARM2440 开发驱动及应用时，不能直接在 HHARM2440 开发板上编辑、编译和调试，而必须把这些工作寄宿到另一台 PC 机上去完成。详细介绍请参见后面章节。

随着微处理器的产生，价格低廉、结构小巧的 CPU 和外设连接提供了稳定可靠的硬件架构，那么限制嵌入式系统发展的瓶颈就突出表现在了软件方面。尽管从八十年代末开始，陆续出现了一些嵌入式操作系统，比较著名的有 Vxwork、pSOS、Neculeus 和 Windows CE。但这些专用操作系统都是商业化产品，其高昂的价格使许多做低端产品的小公司望而却步；而且，源代码封闭性也大大限制了开发者的积极性。另外，结合国内实情，当前国家对自主操作系统的大力支持，也为源码开放的 LINUX 的推广提供了广阔的发展前景。还有，对上层应用开发者而言，嵌入式系统需要的是一套高度简练、界面友善、质量可靠、应用广泛、易开发、多任务，并且价格低廉的操作系统。在不久的将来，从冰箱到收音机都会内置处理器。

因为 Linux 的开放性，许多人认为 Linux 非常适合多数 Internet 设备。他们认为 Linux 可以支持不同的设备，支持不同的配置。Linux 对厂商不偏不倚而且成本极低，能够很快成为用于各种设备的操作系统。如今，业界已经达成共识：即嵌入式 Linux 是大势所趋，其巨大的市场潜力与酝酿的无限商机必然会吸引众多的厂商进入这一领域。

嵌入式操作系统主要有 Palm OS，Windows CE，EPOC，LinuxCE，QNX，ECOS，LYNX，高端嵌入式系统要求许多高级的功能，如图形用户界面和网络支持。很多高端 RTOS 供应商已经提供了这些功能，但其价格也很高端，一般人难以接受。微软的 Windows CE 也有此类功能，却不具备大多数嵌入式系统要求的实时性能，而且难以移植，也曾经有人想以 DOS 为基础用单独的第三方工具拼凑一个系统，但这种努力将是白费。现在需要的是一个便宜、成熟并且提供高端嵌入式系统所必须特性的操作系统，嵌入式 Linux 操作系统以价格低廉、功能强大又易于移植而正在被广泛采用，成为新兴的力量，所以，众多商家纷纷转向了嵌入式 Linux。

Linux 为嵌入操作系统提供了一个极有吸引力的选择，它是个和 Unix 相似、

以核心为基础的、完全内存保护、多任务多进程的操作系统。支持广泛的计算机硬件,包括 MOTOROLA，X86，Alpha，Sparc，MIPS，PPC，ARM，NEC 等现有的大部分芯片。软件源码全部公开，任何人可以修改并在 GNU 通用公共许可证(GNU General Public License)下发行,这样，开发人员可以对操作系统进行定制，再也不必担心像 Microsoft Windows 操作系统中“ 后门 ”的威胁。同时由于有 GPL 的控制,大家开发的东西大都相互兼容，不会走向分裂之路。Linux 用户遇到问题时可以通过 Internet 向网上成千上万的 Linux 开发者请教,这使最困难的问题也有办法解决。Linux 带有 Unix 用户熟悉的完善的开发工具，几乎所有的 Unix 系统的应用软件都已移植到了 Linux 上。Linux 还提供了强大的网络功能，有多种可选择窗口管理器（X windows）。其强大的语言编译器 gcc、g++等也可以很容易得到。不但成熟完善、而且使用方便。

嵌入式系统选择 Linux 的原因：

可应用于多种硬件平台。Linux 已经被移植到多种硬件平台，这对受开销、时间限制的研究与开发项目是很有吸引力的。原型可以在标准平台上开发然后移植到具体的硬件上，加快了软件与硬件的开发过程。

Linux 可以随意地配置不需要任何的许可证或商家的合作关系。唯一的限制是开发者必须做出对 Linux 社区有益的改动。

它是免费的，源代码可以得到。这是最吸引人的。毫无疑问，这会节省大量的开发费用。

微内核直接提供网络支持，而不必象其他操作系统要外挂 TCP/IP 协议包。

Linux 的高度模块化使添加部件非常容易。

Linux 在台式机上的成功，也保证了 Linux 在嵌入式系统中的辉煌前景。

Linux 是一种很受欢迎的操作系统，它与 UNIX 系统兼容，开放源代码。它原本被设计为桌面系统，现在广泛应用于服务器领域。而更大的影响在于它正逐渐的应用于嵌入式系统领域。

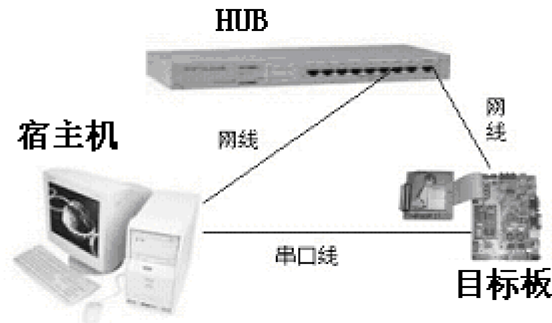
Linux 是一个自由开放的世界，在 Linux（无论 PC 还是嵌入式系统）上进行软件开发都可以在广袤的网络资源中获取帮助。

3.1 使用说明

3.1.1 建立宿主机开发环境

绝大多数的 Linux 软件开发都是以 native 方式进行的，即本机（HOST）开发、调试，本机运行的方式。这种方式通常不适合于嵌入式系统的软件开发，因为对于嵌入式系统的开发，没有足够的资源在本机（即开发板上系统）运行开发

工具和调试工具。通常的嵌入式系统的软件开发采用一种交叉编译调试的方式。交叉编译调试环境建立在宿主机（即一台 PC 机）上，对应的开发板叫做目标板。



开发时使用宿主机上的交叉编译、汇编及连接工具形成可执行的二进制代码，（这种可执行代码并不能在宿主机上执行，而只能在目标板上执行。）然后把可执行文件下载到目标机上运行。调试时的方法很多，可以使用串口，以太网口等，具体使用哪种调试方法可以根据目标机处理器所提供的支持作出选择。宿主机和目标板的处理器一般都不相同，宿主机为 Intel 或 AMD 处理器， 而目标板如 HHARM2440 为 SAMSUNG S3C2440，GNU 编译器提供这样的功能，在编译编译器时可以选择开发所需的宿主机和目标机从而建立开发环境。所以在进行嵌入式开发前第一步的工作就是要安装一台装有指定操作系统的 PC 机作宿主开发机，宿主机上的操作系统一般要求安装 Linux，但 Linux 有多个发行版本，在此，华恒推荐使用 Redhat 9.0 作为本套开发系统的宿主机 PC 操作系统。然后在宿主机上要建立交叉编译调试的开发环境。环境的建立需要许多的软件模块协同工作，这将是一个比较繁杂的工作，但现在只要安装华恒提供的光盘，开发软件包及 GNU 编译工具已完全自动完成了。

当开发环境安装完毕后，会在根目录下生成两个目录：工作目录/HHARM2440 和交叉编译环境目录/opt/host/armv4l。

嵌入式开发通常要求宿主机配置有网络，支持 NFS（为交叉开发时 mount 所用），支持 TFTP 服务器（为下载烧写所用）等等，这个将在后面介绍。

操作系统的选择和安装

我们建议您完全安装的 Redhat9.0 Linux 操作系统，可以使用光盘启动安装 Redhat9.0 Linux 时，刚开始安装不久，安装向导会弹出对话框询问您安装服务器或工作站等，请选择自定义(**Custom**)；安装过程中可以指定 PC 机上网卡的 IP 地址，由于我们的开发套件在烧写时默认的 IP 为 192.168.2.X，所以建议您的 PC 机也在此网段(192.168.2.X)，IP 地址可以在安装时指定，也可以在 PC 机安装好以后指定 IP；在配置防火墙（**Firewall**）时，选择不安装防火墙（**No Firewall**），在选择软件 Package 时选择最后一项：**Everything**，即完全安装。完全安装完以后，大概占用 4.8GB 的硬盘空间。

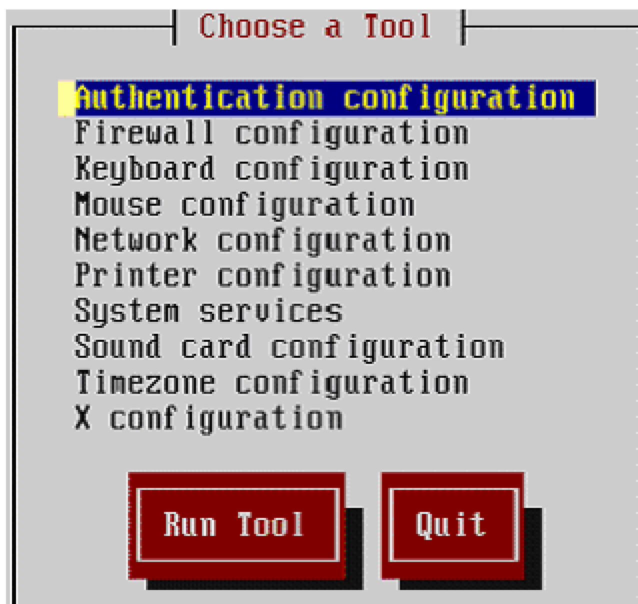
最后会让您选择 Linux 启动以后进入 X 模式还是文本模式，根据自己的爱好决定，进入文本模式时启动花费的时间少一些，即使进入 X windows，也可以按 Ctrl + Alt +Fx(x 在默认情况可以是 1，2，……6)，输入安装 RedHat 时节输入的用户名（root 或已经建立的其它用户）和密码，即可进入操作系统的 Shell 提示符，例如像：[root@localhost root]#

『说明』:安装完成以后也可以修改/etc/inittab 文件来选择进入 X windows 还是进入字符模式。

NFS 和 TFTP 服务器的配置

(1) NFS 的配置：

首先在 REDHAT LINUX PC 机上 shell 提示符[root@….]#执行 setup，弹出菜单界面后:



选中：System services，回车进入系统服务选项菜单，在其中选中 [*]nfs ，然后退出 setup 界面返回到命令提示符下。

vim /etc/exports

将这个默认的空文件修改为只有如下一行内容：

/ (rw) //即根目录可读写，/和(rw)之间要要留空格

然后保存退出（:wq)，然后执行如下命令：

运行以下命令启动 NFS 服务。

/etc/rc.d/init.d/nfs restart 或 service nfs restart

Shutting down NFS mountd: [OK]
Shutting down NFS daemon: [OK]

Shutting down NFS quotas: [OK]
Shutting down NFS services: [OK]
Starting NFS services: [OK]
Starting NFS quotas: [OK]
Starting NFS daemon: [OK]
Starting NFS mountd: [OK]

这样就一切 OK 了！

【注意】

启动完成后，可用如下办法简单测试一下 NFS 是否配置好了：

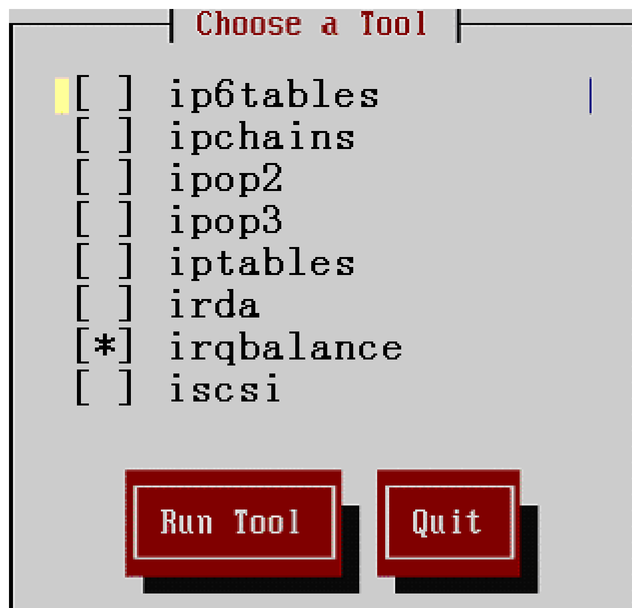
PC 机自己 mount 自己，看是否成功就可以判断 NFS 是否配好了。例如在 PC 机的根目录下执行：(假定 PC 机的 IP 是 192.168.2.126)

mount 192.168.2.126:/ /mnt

然后到/mnt/目录下看是否可以列出所指定的 IP 的机器（可以是本机，当然可以测试其它机器是否可以被 mount）根目录（/）下的所有文件和目录，可以则说明 mount 成功，NFS 配置成功。

(2) TFTP 服务的配置：

在 PC 机上执行 setup，选择 System services，将其中的 tftp 一项选中（出现 [*]表示选中），并去掉 ipchains 和 iptables 两项服务（即去掉它们前面的*号）。



最后，退出 setup，执行如下命令以启动 TFTP 服务：

service xinetd restart

配置完成后，建议简单测试一下 TFTP 服务器是否可用，即自己 tftp 自己，例如在

PC 机上执行：

cd /

cp /etc/inittab /tftpboot/ /*随便拷贝一个文件到/tftpboot 目录下以供下面使用 tftp 命令下载，如果在/tftpboot 目录下没有下面使用 get 命令下载的文件，会提示您没有找到相关文件。*/

tftp 192.168.2.126

tftp> get inittab

若出现如下信息：

Received 741512 bytes in 0.7 seconds

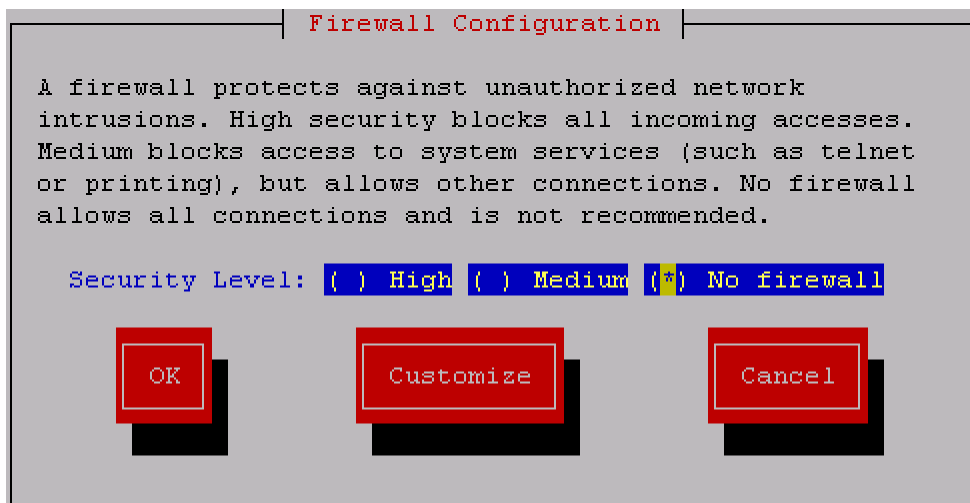
就表示 TFTP 服务器配置成功了。在根目录下就会在刚才下载的 inittab 文件存在了；若弹出信息说：Timed out，则表明未成功，或者用如下命令查看 tftp 服务是否开通：

netstat -a|grep tftp

若 TFTP 服务器没有配置成功，需要按照上述步骤重新检查一遍。

(3) 关闭防火墙：

键入 setup，选择 Firewall configuration，选中 No firewall 关闭防火墙。



『说明』

1. REDHAT LINUX PC 自己 mount 自己成功也不能完全说明 NFS 就可以工作了，因为还有一个防火墙的问题，一般的我们建议客户在安装 REDHAT LINUX 时就默认选择 NO FIREWALL，但若客户没有这样选择的话，REDHAT 启动时会加载防火墙规则，这样你自己能 mount 自己，但其它 PC 和开发板也无法 mount 这台 PC。

注意！setup 里面的防火墙显示永远都是 HIGH，这个是 REDHAT 一直的一个小 BUG，即使你安装时默认选择了 NO FIREWALL，setup 里面也照样会显示防火墙设置是 HIGH 的，这个可以不必理会。只要你选择了一次 NO FIREWALL 就可以了。另

外，建议对 LINUX 操作不是非常熟悉的客户，务必请阅读我们手册附录的“LINUX 常用命令”。

2.有时若根目录不让读写,在 PC 机的命令提示符下执行 chmod 777 / 试试 。

3.REDHAT 在加入网关设置后,网络建立 TCP 链接非常慢,例如 FTP/TELNET/NFS 等都是，建立链接后，以太网通信还是很快的。若您安装网络时加入了网关，就会出现你在开发板 NFS mount REDHAT PC 时，敲入 mount -o nolock pcip:/ /mnt 后非常长的时间不返回 SHELL 提示符，就一直停在那里，这就是建立连接的时间非常的长。一般的，遇到这种情况，建议在 REDHAT LINUX PC 上执行

route del default

即去掉网关，然后再进行 NFS mount 等操作就会非常快了。

4. 测试 NFS 你必须用以太网将开发板和 REDHAT LINUX PC 连接起来才行，连接的方式有两种：一是开发板和 PC 都用普通的网线接到 HUB 或者交换机上；二是用交叉网线将开发板和 PC 直连起来，注意 !这种网线是特制的，内部收发交换的！(一头按普通网线的做法，另一头把 1,3;2,6 交换一下)，测试开发板和 PC 网络是否连通的方式是：开发板正常启动 LINUX 后，在 minicom 里面 ping 一下 PC 看是否能通即可，当然了，用 PC 来 ping 开发板看是否通也是可以的。 注意：开发板必须启动到 LINUX 后才能 ping 通，开发板处在 bootloader 阶段一般是无法 ping 通的，即使这时开发板的 bootloader 初始化了以太网也不行。一般情况下，如果不设置网关，使用 ping 命令时，需要开发板和 PC 机在同一个网段。

若开发板跟 PC 的网络不通，mount 报错信息为：

```
# mount -o nolock 192.168.2.126:/    /mnt
mount: RPC: Unable to receive; eNrno = No route Fo host
mount: Sfsmount failed::Bad file descriptor
mount program didn't pass remote address!
mount: Mounting 192.168.2.126:/ on /mnt failed: Invalid argument
```

6. 关于 tftp 服务安装的问题：有些客户在安装 REDHAT LINUX 时，有时没有选 Custom-->Packages 里面选择 everything，导致没有安装 tftp 服务（典型的就是安装完成后没有/tftpboot 目录，setup-->system services 里面也没有 tftp 选项 ），这时若要重新完全安装就太麻烦了，可如下单独安装该服务：

安装 tftp 的方法：rpm -ivh foo-1.0-2.i386.rpm

其中 foo-1.0-2.i386.rpm 在 Redhat9 的第 3 张光盘里面

『说明』

上面已经把 PC 机上的 Linux 安装和配置好了，下面就可以把华恒提供的开发板的源代码包安装到 PC 机上了。

如果没有配置好上面的 TFTP 服务，HHARM2440 开发板就没有办法使用 tftp 通过以太网下载映像文件。

如果上面的 NFS 没有配置好，则下面开发的应用程序就没有办法使用 NFS 服务，这一项不是必须的，但有这一项服务，可以大大方便调试应用程序的开发。

3.1.2 安装光盘及生成的目录介绍

A. 光盘目录介绍

华恒提供的光盘一般会包含如下目录：

(1)hharm2440-Rx.tgz（Rx 表示版本号）：整个软件源代码和编译器的压缩包。

(2) arminst：安装脚本文件，可以在 shell 提示符下键入 vi 来打开，进入光盘目录以后，键入./arminst,就会提示您安装上面提到的 tgz 扩展名的文件，并且安装相应的编译器和拷贝 minicom 的设置文件。

(3) sjf2440_Rev02：在 windows 2000 终端中进行配置和 u-boot 烧写目录。

(4)DOC：HHARM2440 tech manual-vXX 开发套件的操作及说明手册及其它芯片手册。

(5) Circuit……：开发套件的底板的 PCB 图和原理图，其它的电路图。

其它工具目录：

(6) Linuxconf-1.25r7-3.i386.rpm：linuxconf 的 RPM 包，因为在 Redhat 9.0 上面不能在 shell 提示符下执行 linuxconf 命令,在 Redhat7.2 可以执行 linuxconf，所在在我们以前的手册中写的文档是基于 Redhat7.2 版本的，若现有 Redhat 9.0 如果仍想用 linuxconf 命令执行相关的配置，就要安装此 rpm 包能使用，不过现在的很多配置已经可以不用 linuxconf 也可以配置了，例如：NFS 和防火墙的配置，前面已经有配置方法，如果您仍不知道怎么配置，请注意我们网站上的“常见问题解答”：

http://www.hhcn.com/chinese/embedlinux-res.html

(7) cce.rpm:在字符模式下显示中文的 rpm 包,安装后相当在在 DOS 下面装了 UCDOS，有时我们的安装文件会写一些中文，当执行文件时，若要显示的是中文信息，则看到的是一堆乱码。所以使用了此包，可以看见中文信息。

B．安装开发环境软件包：

请您启动 PC 上的 Redhat 9.0 Linux 操作系统，并将产品附带的光盘插入光盘驱动器，然后执行以下命令：

mount /dev/cdrom /mnt //挂载光盘

cd /mnt

./arminst //执行安装程序

安装的过程中会显示一些提示信息。通常情况下，您只需按下 y 键后，按回车键即可完成整个开发环境的安装。

【注意】

若在文本模下安装时提示中文提示信息，若没有中文环境，会看到一些乱码，用户只需按下 y 回车即可完成全部安装。若要在文本模式下看到中文提示信息，可以安装光盘中带的 cce.rpm 或其它的中文环境即可。

执行完毕后，会在根目录下生成工作目录：**/HHARM2440**，内含 LINUX 内核、应用程序源代码以及各个工具软件。

安装完光盘提供的源代文件和交叉编译环境以后，执行

cd / //回到其它目录才能卸载光盘

umount /mnt //卸载光盘

现在可以取出光盘了。

C．安装光盘后的目录介绍

安装华恒提供的光盘以后，会在您的 PC 的根目录下生成" HHARM2440 "字样的目录，并且把编译器安装到指定的路径中去。

/HHARM2440/u-boot-1.1.1/ bootloader 源码目录，在该目录下执行./HHTECH.mk.u-boot 即可生成 HHARM2440 的 bootloader — u-boot.bin，可以通过修改这些源码来修改 bootloader；

/HHARM2440/linux-2.4.20/：Linux 内核源代码目录及驱动源代码;

/HHARM2440/applications/： 应用程序目录，用户可参考它在其中添加自己的应用;

/HHARM2440/images/： 其下是编译好的映像文件或者可执行文件，其中：zImage 是编译好的 Linux 内核映像文件，u-boot.bin 是编译好的引导程序二进制代码，ramdisk.image.gz 是 ramdisk 文件系统压缩的映像映像文件。

/HHARM2440/modules.TestApp/： 存放 **HHARM2440** 的一些模块程序和测试外转接口的应用程序，例如像摄像头的驱动及测试程序等等。

/HHARM2440/opt.tgz: *编译器的压缩包。*

minirc.dfl：华恒开发板启动时默认的串口终端配置,使用第一个串口(ttyS0)。如果您的串口设置出了问题,可以把此文件拷贝到 PC 机的/etc 目录,然后退出 minicom，再重新进入 minicom 即可。

3.1.3 内核编译

安装华恒提供的光盘时，嵌入式 Linux 内核及设备驱动源代码（光盘安装后建立完备的开发环境）被安装到/HHARM2440/linux-2.4.20 目录下，交叉编译的工具被放置到/opt/host/armv4l 目录下。

GNU 工具集		
armv4l-unknown-linux-gcc	armv4l-unknown-linux-objcopy	rmv4l-unknown-linux-gdb
armv4l-unknown-linux-as	armv4l-unknown-linux-objdump	Armv4l-unknown-linux-gasp
armv4l-unknown-linux-ld	armv4l-unknown-linux-strip	Armv4l-unknown-linux-size
armv4l-unknown-linux-g++	armv4l-unknown-linux-nm	Armv4l-unknown-linux-addr2line
armv4l-unknown-linux-cc1	armv4l-unknown-linux-ar	
armv4l-unknown-linux-cpp	armv4l-unknown-linux-ranlib	
armv4l-unknown-linux-cc1plus	armv4l-unknown-linux-strings	

cd /HHARM2440/linux-2.4.20

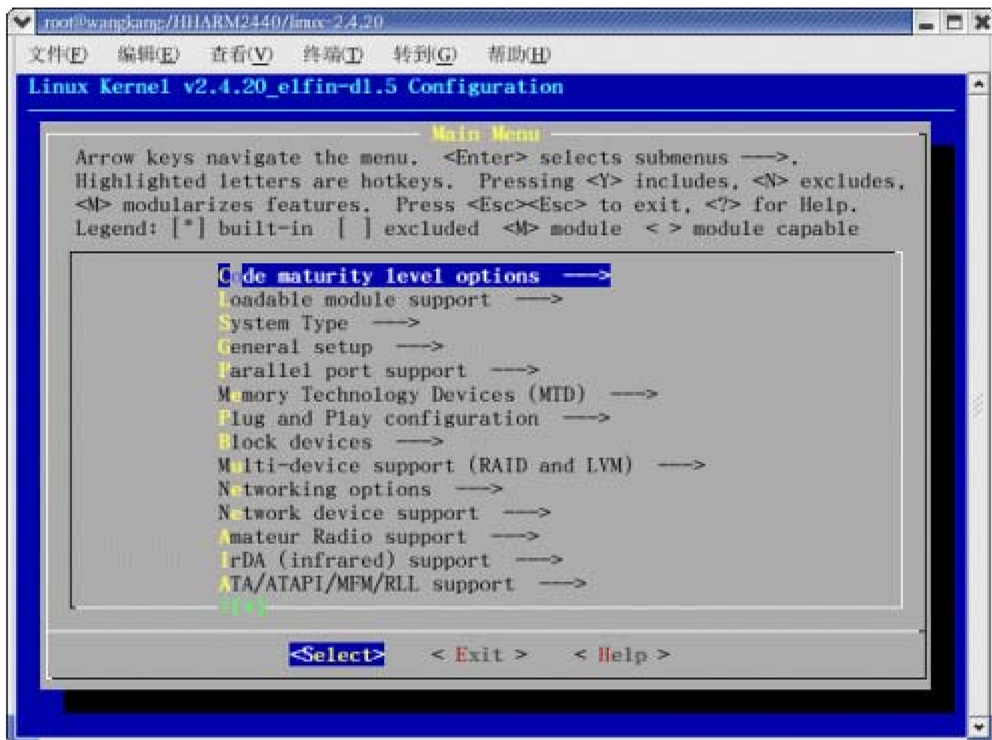
make zImage

编译完成后，就会自动把 zImage 复制到/tftpboot/目录下以供 TFTP 下载。

如果需要对内核进行配置和裁剪，可以在/HHARM2440/linux-2.4.20 目录下键入

make menuconfig

则出现如下界面，可逐项对内核和驱动模块进行选择和配置，可见内核版本为：Linux Kernel v2.4.20_elfin-d1.5。



完成自己的设置后，退出，保存配置，然后键入 make zImage 编译自己定制的内核，并生成映像文件 zImage。如果想了解编译的过程，可键入如下命令把输出结果重定向到 log 文件，进行观察：

make zImage &>log **//把编译信息输出到文件 log 中**

vim log

即可看到完整的编译过程，下面摘录部分编译过程，分析如下：

…… ……

/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T arch/arm/vmlinux.lds arch/arm/kernel/head-armv.o arch/arm/kernel/init_task.o init/main.o init/version.o init/do_mounts.o \

--start-group \

arch/arm/kernel/kernel.o arch/arm/mm/mm.o arch/arm/mach-s3c2440/s3c2440.o kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \

drivers/char/char.o drivers/block/block.o drivers/misc/misc.o drivers/net/net.o drivers/base/base.o drivers/base/fs/base-fs.o drivers/media/media.o drivers/serial/serial.o drivers/ide/idedriver.o drivers/scsi/scsidrv.o drivers/sound/sounddrivers.o drivers/mtd/mtdlink.o drivers/video/video.o drivers/usb/usbdrv.o drivers/usb/device/usbd_drv.o \

net/network.o \

arch/arm/nwfpe/math-emu.o arch/arm/lib/lib.a /HHARM2440/linux-2.4.20/lib/lib.a \
--end-group \
-o vmlinux

////////////////////////////////////// 以下为插入的一段说明 //////////////////////////////////////

这里生成的就是 linux-2.4.20 目录下的 **vmlinux** 文件。

看看它内部的段结构和地址：

[root@wangkang linux-2.4.20]# /opt/host/armv4l/bin/armv4l-unknown-linux-objdump --header vmlinux

```
vmlinux:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00014000  c0008000  c0008000  00008000  2**5
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         001c99f4  c001c000  c001c000  0001c000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .kstrtab      00006363  c01e59f4  c01e59f4  001e59f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 __ex_table    000009f0  c01ebd60  c01ebd60  001ebd60  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  4 __ksymtab     00002d88  c01ec750  c01ec750  001ec750  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .data         00015b68  c01f0000  c01f0000  001f0000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  6 .bss          00036280  c0205b80  c0205b80  00205b80  2**5
                  ALLOC
  7 .stab         005df234  00000000  00000000  00205b80  2**2
                  CONTENTS, READONLY, DEBUGGING
  8 .stabstr      001b83d1  00000000  00000000  007e4db4  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .comment      00005cf8  00000000  00000000  0099d185  2**0
                  CONTENTS, READONLY
```

[root@wangkang linux-2.4.20]# 它的地址分布就是由 /HHARM2440/linux-2.4.20/arch/arm/vmlinux.lds 文件指定的：

OUTPUT_ARCH(arm)
ENTRY(stext)

```
SECTIONS
{
	. = 0xc0008000;
	.init : {			/* Init code and data		*/
		_stext = .;
		__init_begin = .;
			*(.text.init)
		__proc_info_begin = .;
			*(.proc.info)
		__proc_info_end = .;
		__arch_info_begin = .;
			*(.arch.info)
		__arch_info_end = .;
		__tagtable_begin = .;
			*(.taglist)
		__tagtable_end = .;
			*(.data.init)
		. = ALIGN(16);
		__setup_start = .;
			*(.setup.init)
		__setup_end = .;
		__initcall_start = .;
			*(.initcall.init)
		__initcall_end = .;
		. = ALIGN(4096);
		__init_end = .;
	}

	/DISCARD/ : {			/* Exit code and data		*/
		*(.text.exit)
		*(.data.exit)
		*(.exitcall.exit)
	}

	.text : {			/* Real text segment		*/
		_text = .;		/* Text and read-only data	*/
			*(.text)
```

```
                *(.fixup)
                *(.gnu.warning)
                *(.rodata)
                *(.rodata.*)
                *(.glue_7)
                *(.glue_7t)
        *(.got)                 /* Global offset table          */

        _etext = .;             /* End of text section          */
}

.kstrtab : { *(.kstrtab) }

. = ALIGN(16);
__ex_table : {                  /* Exception table              */
        __start___ex_table = .;
                *(__ex_table)
        __stop___ex_table = .;
}

__ksymtab : {                   /* Kernel symbol table          */
        __start___ksymtab = .;
                *(__ksymtab)
        __stop___ksymtab = .;
}

. = ALIGN(8192);

.data : {
        /*
          * first, the init task union, aligned
          * to an 8192 byte boundary.
          */
        *(.init.task)

        /*
          * then the cacheline aligned data
```

```
         */
        . = ALIGN(32);
        *(.data.cacheline_aligned)

        /*
         * and the usual data section
         */
        *(.data)
        CONSTRUCTORS

        _edata = .;
    }

    .bss : {
        __bss_start = .;        /* BSS                  */
        *(.bss)
        *(COMMON)
        _end = . ;
    }
                        /* Stabs debugging sections.       */
    .stab 0 : { *(.stab) }
    .stabstr 0 : { *(.stabstr) }
    .stab.excl 0 : { *(.stab.excl) }
    .stab.exclstr 0 : { *(.stab.exclstr) }
    .stab.index 0 : { *(.stab.index) }
    .stab.indexstr 0 : { *(.stab.indexstr) }
    .comment 0 : { *(.comment) }
```

// 以上为插入的一段说明 //

紧接着的这一句说明了 **System.map** 文件是如何生成的。

```
/opt/host/armv4l/bin/armv4l-unknown-linux-nm vmlinux | grep -v '\(compiled\)\|\(\.o$\)\|\( [aUw] \)\|\(\.\.ng$\)\|\(LASH[RL]DI\)' | sort > System.map
```

上面是真正运行的 LINUX 内核，下面则是将这个内核用 gzip 进行压缩后，与 head.S 这个解压缩代码部分整合到一起生成 zImage：

```
make[1]: Entering directory `/HHARM2440/linux-2.4.20/arch/arm/boot'
make[2]: Entering directory `/HHARM2440/linux-2.4.20/arch/arm/boot/compressed'
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__ASSEMBLY__ -D__KERNEL__
-I/HHARM2440/linux-2.4.20/include -mapcs-32 -D__LINUX_ARM_ARCH__=4
```

-march=armv4 -msoft-float -traditional -c head.S
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__KERNEL__ -I/HHARM2440/linux-2.4.20/include -O2 -DSTDC_HEADERS -mapcs-32 -D__LINUX_ARM_ARCH__=4 -march=armv4 -mtune=arm9tdmi -mshort-load-bytes -msoft-float -fpic -Uarm -D__KERNEL__ -I/HHARM2440/linux-2.4.20/include -c -o misc.o misc.c

//请注意下面的几句话：

//先将 ELF 格式的 vmlinux 转换为二进制格式的，尺寸大大缩水，由原来
//的 2M 多变为 700～800K 字节，并用 piggy 作为临时文件名

/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S /HHARM2440/linux-2.4.20/vmlinux piggy

//对内核进行 gzip 压缩

gzip -9 < piggy > piggy.gz

//再转换为.o 格式的文件

/opt/host/armv4l/bin/armv4l-unknown-linux-ld -r -o piggy.o -b binary piggy.gz
rm -f piggy piggy.gz

//将解压缩部分代码和压缩内核链接为一个新的 vmlinux

/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T vmlinux.lds head.o misc.o piggy.o /opt/host/armv4l/lib/gcc-lib/armv4l-unknown-linux/2.95.2/soft-float/libgcc.a -o vmlinux

//再次将解压缩代码部分的 ELF 格式转换为二进制文件格式，即最终的
//zImage 文件，并复制到/tftpboot 目录下以供 TFTP 下载烧写。

make[2]: Leaving directory `/HHARM2440/linux-2.4.20/arch/arm/boot/compressed'
/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S compressed/vmlinux zImage
cp zImage /tftpboot -f
make[1]: Leaving directory `/HHARM2440/linux-2.4.20/arch/arm/boot'

3.1.4 利用 u-boot 烧制内核映像和文件系统

客户拿到手的开发板，我们已经把相关代码烧写进去了，开发板已经可以启动了，若对内核和文件系统做出修改，则需要重新烧写 Linux 内核映像文件 zImage 和 ramdisk 文件系统文件 ramdisk.image.gz，但为了不至于频繁的烧写内核和文件系统，可以把内核和文件系统先下载至 SDRAM 中直接启动（而不是从 flash 读到 SDRAM 中运行），测试一下新的内核和文件系统是否可以正常使用，如果确定最终需要烧写的文件，再进行烧写也不迟。这一功能可以极大的方便内核调试。内

核和 ramdisk 映象下载到内存中后，可以使用 go 命令直接启动刚下载的内核。其使用步骤如下：

SDMK2440# tftp 30008000 zImage
SMDK2440# tftp 30800000 ramdisk.image.gz
SMDK2440# bootm

即可直接启动刚下载的内核，并最终进入 shell 提示符“ # ”。

这样做的的好处是不用频繁的烧写 flash，可以延长 flash 的使用寿命，推荐用户在调试自己编译的内核、加入自己的应用程序等的时候使用此方法。等确定最终烧写的文件时，再烧写到 flash 中去。

烧写的大致过程如下：

1．按复位键重启开发板，在 minicom 中应该有启动信息,立即按空格键（而不是回车键），让开发板进入 u-boot 的提示符” SMDK2440 #”，进行以下烧写；

2．下载、烧写内核 zImage；

3．下载、烧写文件系统 ramdisk.image.gz；

4．若有其它文件系统，下载、烧写其它文件系统。

◆※注释※◆

minicom: linux 下的一个终端程序，就像 windows 中的超级终端一样，只需要在 Shell 提示”#”下键入 minicom 就可以进入 minicom 终端了，更改相应的设置可以按 Ctrl+a，o 进入配置， 退出 minicom 请按 Ctrl +a,q，详细说明，请看 minicom 终端界面的最下面一行提示，寻求有关帮助，在本手册的附录部分也有介绍。

◆※注释※◆

详细下载及烧写命令如下：

tftp 0x30008000 zImage //通过 TFTP 下载内核映像文件
fl 0x20000 0x30008000 0xf0000 //烧写刚下载的文件到指定的位置
tftp 0x30800000 ramdisk.image.gz //下载压缩的 ramdisk 文件系统映像文件
fl 0x200000 0x30800000 0x300000
tftp 0x30008000 cramfs.img //下载 cramfs 文件系统映像文件
fl 0x600000 0x30008000 xxxx //烧写 cramfs 文件系统，其中 xxxx 要替换成十六进制数值
tftp 0x30008000 jffs2.img //下载 jffs2 文件系统映像文件
fl 0xa00000 0x30008000 xxxx //烧写

『说明』

A.下载 linux 内核和文件系统映像文件都是通过 tftp 服务来完成的，宿主机必

须把 TFTP 服务器配置好，相关配置，请见前面有关章节的说明。ftp 命令默认从 TFTP 服务器的/tftpboot 目录里下载文件到开发板的 RAM 中的，要保证所指定的 TFTP 服务器的/tftpboot 目录下有被下载的文件，否则下载时会提示您“File not Found”，可以把 HHARM2440/images 目录下我们出厂时的文件拷贝到 TFTP 服务器的/tftpboot 目录下以供 tftp 下载。

B. fl 命令后的三个参数的意义为：第一个参数是烧写到 Flash 的地址；第二个参数是刚才通过 tftp 把文件下载到开发板中 SDRAM 中的地址；最后一个参数表示在烧写指定的文件时，在 flash 中给这个文件的空间大小，请注意最后一个参数的值一定要比刚才通过 tftp 命令下载的文件的大小要大，否则，刚才的文件只烧了部分，没有烧写完全，此值是由使用 tftp 命令下载的文件的大小确实的，在通过 tftp 命令下载的时候，最后会提示您（十六进制值的）文件的大小，保持最后 4 个数字为“0”就可以了。

C. tftp 命令和 fl 命令后的所有十六进制数值在键入命令时，0x 前缀可以省略。

D.首先通过以太网从指定的 TFTP 服务器上下载指定的文件到内存中指定的区域，然后把 SDRAM 中指定区域中的内容烧写到 flash 中指定的位置。

E.如果把开发板和PC机的以太网口直接相连，要有交叉对接网线(1,3；2,6交换)，如果要用普通的网线，则要经过网络连接设备(集线器，交换机等等)。建议用对接网线。

F．u-boot的提示符(=>)下，若已经键入过命令，按回车表示确认键入的命令，若再次按下回车，则重复行此命令。

G．在u-boot提示符下键入help查看fl等命令的帮助。

H．在/HHARM2440/images/HHTECH-burn-cmd文件中，保存了下载和烧写时的命令，以方便烧写时使用“粘贴”(鼠标在要选择的命令中快速按三下或拖动)，“复制”(在minicom中右击鼠标)。

I．在u-boot提示符下，多个命令可以写到一行上，各个命令之间使用分号间隔。

烧写过程出现的信息如下：

U-Boot 1.1.1 (Sep 21 2005 - 14:21:03)

U-Boot code: 32F00000 -> 32F161F4 BSS: -> 32F19650
RAM Configuration:
Bank #0: 30000000 64 MB
Flash Memory Start 0x0
Device ID of the Flash is 18
Flash: 16 MB
In: serial

Out: serial
Err: serial
Autoboot is in process. Press Space to stop Autobooting...
SMDK2440 # //按空格键，让其停留在“ SMDK2440 ”提示符下，如果操作慢了，请按一下 HHARM2440 底板上的复位键，重复这一过程。

A．设置 TFTP 服务器的 IP 地址。

SMDK2440 # **setenv serverip 192.168.2.126**

//设置 TFTP 服务器的地址，如果不设置，开发板就会使用 u-boot 中默认的 IP 192.168.2.126，当然，如果您的 PC 机的 IP 正好是 u-boot 默认要求的 IP 地址，那就不用执行此命令。

B．通过 TFTP 下载 Linux 内核映像文件 zImage

SMDK2440 # **tftp 30008000 zImage**
NetLoop 1
NetLoop 2
<DM9000> I/O: 8000300, VID: 90000a46
NetLoop 3
NetLoop 4
NetOurIP =c0a80278
NetServerIP = c0a8027e //// 【注意】这里 c0a8027e 就是 192.168.2.126，即要求 LINUX PC 的 IP 地址必须为 192.168.2.126，这个地址是可以修改修改的，参见下面介绍。
NetOurGatewayIP = c0a80201
NetOurSubnetMask = ffffff00
ARP broadcast 1
ARP broadcast 2
ARP broadcast 3
ARP broadcast 4
ARP broadcast 5
TFTP from server 192.168.2.126; our IP address is 192.168.2.120
Filename 'zImage'.
Load address: 0x30008000
Loading:
###
###
###
done

Bytes transferred = 904096 (ecba0 hex)

//【注意】括号中的数值 echa0 表示刚下载文件的大小(十六进制值)，下面使用 fl 命令烧写时最后一个参数的值要比这个值大。
如果 PC 机上 TFTP 服务没有配置正确，或配置的 TFTP 服务器的 IP 地址和开板要求的不一致，会反复出现以下信息提示：

ARP broadcast 1

ARP broadcast 2

ARP broadcast 3

ARP broadcast 4

ARP broadcast 5

C．烧写刚下载的内核映像文件

SMDK2440 # **fl 20000 30008000 f0000** //注意 fl 命令中的几个参数的意义，其中 f0000 是给刚才下的文件在 flash 中的空间，一定要大于上一行中提示的十六进制值且保持最后 4 个数字是 0，f0000 比上一行 ecba0 大。

start_sect=0x1,end_sect=0x7

************erase sector 0x1***********

************erase sector 0x2***********

************erase sector 0x3***********

************erase sector 0x4***********

************erase sector 0x5***********

************erase sector 0x6***********

************erase sector 0x7***********

----------program sector 0x1---------

----------program sector 0x2---------

----------program sector 0x3---------

----------program sector 0x4---------

----------program sector 0x5---------

----------program sector 0x6---------

----------program sector 0x7---------

D．下载压缩过的的文件系统映像文件 ramdisk.image.gz

SMDK2440 # **tftp 30800000 ramdisk.image.gz**

NetLoop 1

NetLoop 2

<DM9000> I/O: 8000300, VID: 90000a46

NetLoop 3

NetLoop 4
NetOurIP =c0a80278
NetServerIP = c0a8027a
NetOurGatewayIP = c0a80201
NetOurSubnetMask = ffffff00
ARP broadcast 1
ARP broadcast 2
TFTP from server 192.168.2.126; our IP address is 192.168.2.120
Filename 'ramdisk.image.gz'.
Load address: 0x30800000
Loading:
###
… … … … … … … … … … … … … … … … … …
###########################
done
Bytes transferred = 2127363 (257603 hex)

E．烧写文件系统映像文件

SMDK2440 # **fl 200000 30800000 260000**
start_sect=0x10,end_sect=0x27
************erase sector 0x10***********
************erase sector 0x11***********
************erase sector 0x12***********
************erase sector 0x13***********
************erase sector 0x14***********
************erase sector 0x15***********
************erase sector 0x16***********
************erase sector 0x17***********
************erase sector 0x18***********
************erase sector 0x19***********
************erase sector 0x1a***********
************erase sector 0x1b***********
************erase sector 0x1c***********
************erase sector 0x1d***********
************erase sector 0x1e***********
************erase sector 0x1f***********
************erase sector 0x20***********

```
************erase sector 0x21***********
************erase sector 0x22***********
************erase sector 0x23***********
************erase sector 0x24***********
************erase sector 0x25***********
************erase sector 0x26***********
************erase sector 0x27***********
----------program sector 0x10---------
----------program sector 0x11---------
----------program sector 0x12---------
----------program sector 0x13---------
----------program sector 0x14---------
----------program sector 0x15---------
----------program sector 0x16---------
----------program sector 0x17---------
----------program sector 0x18---------
----------program sector 0x19---------
----------program sector 0x1a---------
----------program sector 0x1b---------
----------program sector 0x1c---------
----------program sector 0x1d---------
----------program sector 0x1e---------
----------program sector 0x1f---------
----------program sector 0x20---------
----------program sector 0x21---------
----------program sector 0x22---------
----------program sector 0x23---------
----------program sector 0x24---------
----------program sector 0x25---------
----------program sector 0x26---------
----------program sector 0x27---------
```

F.其它文件系统的下载、烧写

```
tftp   30008000   cramfs.img
fl   600000 30008000   xxxx
tftp   30008000   jffs2.img
fl   a00000 30008000   xxxx
```

『说明』

1．后面没有列出 cramfs 和 JFFS2 文件系统烧写的相关信息，和烧写 ramdisk.image.gz 一样，先下载，再烧写，只是烧写到 flash 中的位置不一样，这个位置是根据**/HHARM2440/linux-2.4.20/drivers/mtd/maps/s3c2440.c** 文件中指定的位置确定的，cramfs.img 烧写到 flash 中的 0x600000 位置，共 4MB 空间，jffs2 烧写到 0xa000000，共 6MB 空间。

2．用户可以修改 u-boot 中指定的 TFTP 服务器和开发板的 IP 地址。

修改 TFTP 服务器的 IP 地址有两种方法。

A．不用重新烧写 u-boot.bin，临时修改 TFTP 服务器的 IP 地址。

SMDK2440 # setenv serverip 192.168.2.111

B．修改设置 TFTP 服务器的 IP 源文件，重新烧写 u-boot.bin 文件。

vi /HHARM2440/u-boot-1.1.1/include/configs/smdk2440.h

修改如下一行：

#define CONFIG_SERVERIP 192.168.2.126 //122.2.168.192

然后运行 u-boot-1.1.1 目录下的 HHTECH 字母打头的文件(或执行 make distclean;make smdk2440_config;make 也可)，重新编译 u-boot，重新烧写新的 u-boot.bin，以后，再次下载映像文件时 u-boot 中要求的 TFTP 服务器的 IP 就改成新的 IP 了。

在 u-boot 提示符下键入 printenv 命令可以查看设置的环境变量。

```
SMDK2440 # printenv
bootdelay=3
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
ipaddr=192.168.2.120
gatewayip=192.168.2.1
netmask=255.255.255.0
serverip=192.168.2.126

Environment size: 141/65532 bytes
SMDK2440 #
```

3.1.5 嵌入的 LINUX 系统

烧写完 u-boot，Linux 内核以及 ramdisk 文件系统以后，上电复位或硬件 RESET 的方式重新启动目标板，或者在 u-boot 提示符下输入命令

SMDK2440# **reset**

来启动 Linux。

通过串口终端可以看到开发板上 u-boot 的检测、引导内嵌 Linux 操作系统启动打印到串口的信息和 Linux 的整个启动信息。

U-Boot 1.1.1 (Sep 21 2005 - 14:42:37)

U-Boot code: 32F00000 -> 32F161F4 BSS: -> 32F19650
RAM Configuration:
Bank #0: 30000000 64 MB
Flash Memory Start 0x0
Device ID of the Flash is 18
Flash: 16 MB
In: serial
Out: serial
Err: serial
Autoboot is in process. Press Space to stop Autobooting...
Coping Kernel form Flash to RAM ...
Booting image at 30008000 ...
Uncompressing Linux...Linux version 2.4.20_elfin-d1.5 (root@wangkang) (gcc version 2.95.2 20000516 (r5CPU: ARM/CIRRUS Arm920Tid(wb) revision 0
Machine: Samsung-SMDK2440
On node 0 totalpages: 12288
zone(0): 12288 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: initrd=0x30800000,0x400000 root=/dev/ram mem=48M init=/lin0Console: colour dummy device 80x30
Calibrating delay loop... 151.96 BogoMIPS
Use CONFIG_INSTANT_ON_LPJ=759808 for Instant On.
Memory: 48MB = 48MB total
Memory: 42228KB available (1830K code, 345K data, 80K init)
Dentry cache hash table entries: 8192 (order: 4, 65536 bytes)
Inode cache hash table entries: 4096 (order: 3, 32768 bytes)
Mount-cache hash table entries: 1024 (order: 1, 8192 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 16384 (order: 4, 65536 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4

Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
CPU clock = 304.800000 Mhz, HCLK = 101.600000 Mhz, PCLK = 50.800000 Mhz
PWM-Timers Management Module Loaded.
Disabling the Out Of Memory Killer
Starting kswapd
Journalled Block Device driver loaded
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
NTFS driver v1.1.22 [Flags: R/O]
JFFS version 1.0, (C) 1999, 2000 Axis Communications AB
JFFS2 version 2.1. (C) 2001, 2002 Red Hat, Inc., designed by Axis Communication.LCDCON1 = 779
LCDCON2 = 4fc041
LCDCON3 = 90ef1e
LCDCON4 = d03
LCDCON5 = 16809
Console: switching to colour frame buffer device 30x40
Installed S3C2440 frame buffer
LCDCON1 = 3f00779
LCDCON2 = 4fc041
LCDCON3 = 90ef1e
LCDCON4 = d03
LCDCON5 = 14809
kbd init
#####keyboard init
result is 0
####requesting irq is ok!
pty: 256 Unix98 ptys configured
s3c2440-ts initialized
Uniform Multi-Platform E-IDE driver Revision: 6.31
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
ioremap:c38a8000
ide0: CPCI405 IDE interface
IDE: waiting for drives to settle...

IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
dm9000x probe...
HHTech DM9000 eth0 I/O: c38aa300,VID: 90000a46,MAC: 00:13:F6:6C:87:89:
RAMDISK driver initialized: 16 RAM disks of 400000K size 1024 blocksize
loop: loaded (max 8 devices)
ttyS%d0 at I/O 0x50000000 (irq = 52) is a S3C2440
ttyS%d1 at I/O 0x50004000 (irq = 55) is a S3C2440
ttyS%d2 at I/O 0x50008000 (irq = 58) is a S3C2440
SCSI subsystem driver Revision: 1.00
kmod: failed to exec /sbin/modprobe -s -k scsi_hostadapter, errno = 2
the date write is: aa55
the read data is: aa55
Init uda1380 finished.
UDA1380 audio driver initialized
S3C2440 flash: probing 16-bit flash bus
Using buffer write method
Using static partition definition
Creating 4 MTD partitions on "S3C2440 Flash":
0x00000000-0x00100000 : "bootloader"
0x00100000-0x00200000 : "kernel"
0x00200000-0x00600000 : "cramfs"
0x00600000-0x01000000 : "jfss2"
usb.c: registered new driver usbdevfs
usb.c: registered new driver hub
S3C2440 USBD Controller Core Initialized
USB Function Ethernet Driver Interface
Initializing S3C2440 buffer pool for DMA workaround
usb-ohci.c: USB OHCI at membase 0xe9000000, IRQ 26
usb.c: new USB bus registered, assigned bus number 1
hub.c: USB hub found
hub.c: 2 ports detected
Initializing USB Mass Storage driver...
usb.c: registered new driver usb-storage
USB Mass Storage support registered.
usbdcore: usbdcore 0.1 034 2002-06-12 20:00 (dbg="")

```
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 4096 bind 4096)
opening device
IP-Config: Incomplete network configuration information.
Trying to free IRQ0
Function entered at [<c001ef98>] from [<c00fb91c>]
 r4 = C01FE184
Function entered at [<c00fb8d4>] from [<c0154768>]
 r6 = 00001003  r5 = C0201ED8  r4 = C01FE184
Function entered at [<c01546d4>] from [<c0155e70>]
 r5 = 00001002  r4 = C01FE184
Function entered at [<c0155e10>] from [<c0016404>]
 r7 = C02068A0  r6 = C0202378  r5 = C02FFD00  r4 = 00000000
Function entered at [<c00163a4>] from [<c00169e8>]
 r6 = C02068B8  r5 = C02327BC  r4 = C001B7F4
Function entered at [<c0016908>] from [<c0008728>]
 r8 = C01F3684  r7 = C02068A0  r6 = C02068B8  r5 = C02327BC
 r4 = C001BBC8
Function entered at [<c0008710>] from [<c0008770>]
 r4 = 00000000
Function entered at [<c0008744>] from [<c001c03c>]
Function entered at [<c001c02c>] from [<c001f904>]
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 4096K
EXT2-fs warning: checktime reached, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Mounted devfs on /dev
Freeing init memory: 80K
mount /etc as ramfs
re-create the /etc/mtab entries
reboot(magic=00000000
)...
opening device
```

```
Starting S3C2440 Camera
camif_dst_x = 320, camif_dst_y = 240

[ Camera Preview Test ]


BusyBox v1.00-pre10 (2005.08.31-07:46+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # ls

bin         hhtech      linuxrc     proc        usr
dev         jffs2       lost+found  sbin        var
etc         lib         mnt         tmp
~ # ps
  PID  Uid        VmSize Stat Command
    1 root          524 S    init
    2 root              SW   [keventd]
    3 root              SWN  [ksoftirqd_CPU0]
    4 root              SW   [kswapd]
    5 root              SW   [bdflush]
    6 root              SW   [kupdated]
    9 root              SW   [mtdblockd]
   10 root              SW   [khubd]
   21 root          528 S    inetd
   35 root              SWN  [jffs2_gcd_mtd3]
   40 root          508 S    syslogd -n
   41 root         3024 S    treeview
   43 root          680 S    -sh
   44 root         3024 S    treeview
   45 root         3024 S    treeview
   46 root         3024 S    treeview
   47 root         3024 S    treeview
   49 root          604 R    ps
~ # cat /proc/interrupts
  0:          1    DM9000 device
 10:          0    Timer 0
```

```
 11:          0    Timer 1
 12:          0    Timer 2
 13:          0    Timer 3
 14:      19276    timer
 18:          0    I2SSDI
 19:          0    I2SSDO
 25:          4    USBD
 26:          0    usb-ohci
 27:         17    CAM_IIC
 32:          0    test_keyboard
 52:         38    serial_elfin__rx
 53:        109    serial_elfin__tx
 54:          0    serial_elfin__err
 61:          0    s3c2440-ts
 62:          0    s3c2440-ts
 64:          0    CAM_S
Err:          0
~ #
```

3.1.6 利用 u-boot 更新 u-boot 自身

『说明』：**用户在使用开发板的过程中，若遇到系统崩溃、修改了引导代码，就要考虑重新烧写 u-boot。在没有上述情况时，无须重新烧写 u-boot，开发板在出厂时我们已经烧写了 u-boot。**

『说明』在嵌入式系统中，我们把引导系统的初始化部分的代码统称为 bootloader，相当于 PC 机的 BIOS。在我们公司提供的积各种嵌入式开发平台中（**ARM** 系列套件、**Powr PC** 系列套件，**Coldfire** 系列套件、**ADSP** 系列套件等），有的引导代码用的是 ppcboot，有的是 u-boot，有的是 bootloader 等等，但实际烧写到 flash 中的文件一般为 ppcboot.bin、u-boot.bin、bootloader.bin 等二进制代码文件。

烧写u-boot本身

当需要对u-boot本身进行修改的时候，一种方式是在u-boot引导以后，通过tftp下载后，使用fl进行烧写，实现对自身更新；另一种方式是通过JTAG进行烧写。下面介绍前一种更新u-boot的方式，此种方式是在u-boot可以引导的情况下进行的，下载和烧写就是下载和烧写内核和文件系统映像文件一样，只是烧写的位置不同而已。

最新的u-boot可以保护flash的扇区，我们默认保护第一个扇区——引导代码，也可以保护flash的所有扇区。被保护的flash扇区，在烧写之前，一定要取消保护才能进行擦除烧写。

已经烧写了最新u-boot后的开发板，在能引导的情况下，不能通过JTAG再进入烧写，除非把现在的引导破坏，让开发板引导崩溃才能通过JTAG进行烧写。其实在开发板可以引导的情况下，没有必要使用JTAG对开发板进行烧写，可以通过u-boot更新自身。

SMDK2440 # **flinfo** //打印flash信息

```
Bank # 1: Unknown Vendor intel E28F128J3A150
    Size: 16 MB in 128 Sectors
    Sector Start Addresses:
00000000 (RO) 00020000    00040000    00060000    00080000
000A0000    000C0000    000E0000    00100000    00120000
00140000    00160000    00180000    001A0000    001C0000
001E0000    00200000    00220000    00240000    00260000
00280000    002A0000    002C0000    002E0000    00300000
00320000    00340000    00360000    00380000    003A0000
003C0000    003E0000    00400000    00420000    00440000
00460000    00480000    004A0000    004C0000    004E0000
00500000    00520000    00540000    00560000    00580000
005A0000    005C0000    005E0000    00600000    00620000
00640000    00660000    00680000    006A0000    006C0000
006E0000    00700000    00720000    00740000    00760000
00780000    007A0000    007C0000    007E0000    00800000
00820000    00840000    00860000    00880000    008A0000
008C0000    008E0000    00900000    00920000    00940000
00960000    00980000    009A0000    009C0000    009E0000
00A00000    00A20000    00A40000    00A60000    00A80000
00AA0000    00AC0000    00AE0000    00B00000    00B20000
00B40000    00B60000    00B80000    00BA0000    00BC0000
00BE0000    00C00000    00C20000    00C40000    00C60000
00C80000    00CA0000    00CC0000    00CE0000    00D00000
00D20000    00D40000    00D60000    00D80000    00DA0000
00DC0000    00DE0000    00E00000    00E20000    00E40000
00E60000    00E80000    00EA0000    00EC0000    00EE0000
00F00000    00F20000    00F40000    00F60000    00F80000
```

00FA0000 00FC0000 00FE0000

从上面我们看到flash共有128个128KB（十六进制20000）大小的扇区，第一个扇区已经被保护了00000000 (RO)，如果想重新烧写u-boot，需要打开保护。

SMDK2440 # **protect off 1:0** //去除第一个扇区的保护

Un-Protect Flash Sectors 0-0 in Bank # 1

SMDK2440# tftp 30008000 u-boot.bin //从指定的TFTP服务器上下载u-boot

SMDK2440# fl 0 30008000 20000 //烧写u-boot

如果在没有打开保护的情况下烧写，就会出现以下错误提示信息：

```
SMDK2440 # fl 0 30008000 20000
start_sect=0x0,end_sect=0x0
************erase sector 0x0***********
----------program sector 0x0---------
Error Command Sequence!mflash_program_sector error : status read
flash not completed for error
```

『说明』：保护和去除保护的命令如下：

protect on 1：n-m

protect off 1：n-m

这里的n和m的值为flash芯片128个扇区中整数(0,1,2,…… 126,127),n < m 。参数m可以省略，省略m表示对指定的扇区执行操作，1是protect函数执行的判断条件，如果是0就不执行相关操作。

3.1.7 利用 JTAG 烧写引导程序 u-boot

HHARM2440 目前只能在 windows 操作系统环境下进行引导代码的烧写，下面以 Windows 2000 Professional 为例进行介绍。

A．安装使用 JTAG 时的端口驱动。

B．安装烧写 flash 时用到的 SJF2440。

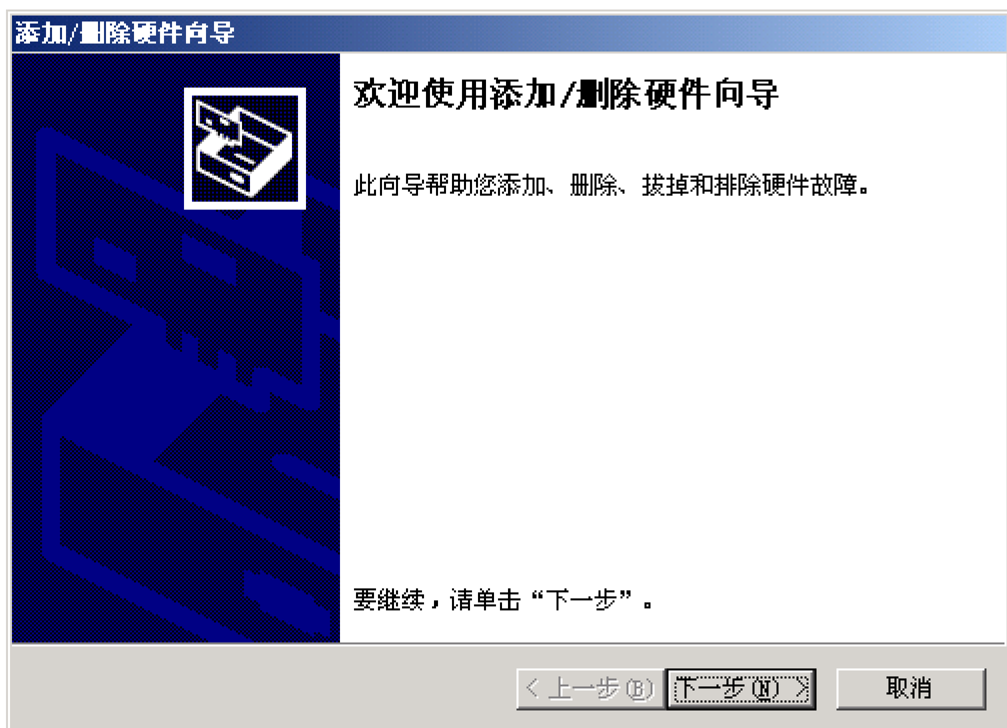
C．通过 JTAG 把引导代码烧写到 flash 中去。

详细过程如下：

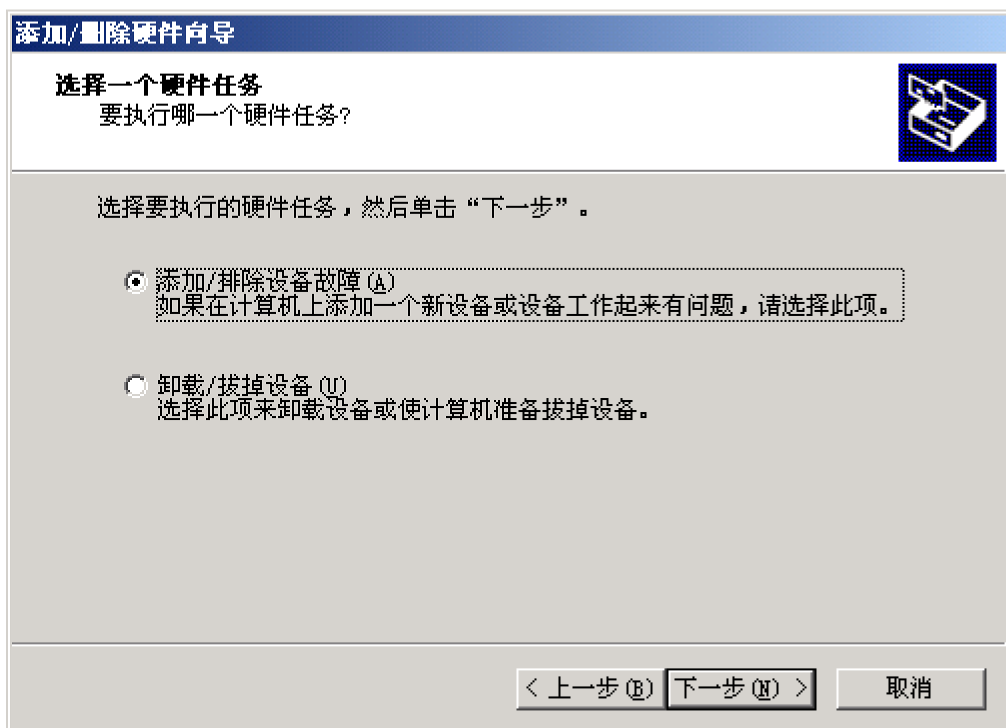
1．安装 Jtag 驱

为了方便，我们直接把 sjf2440_Rev02 目录放在 D 盘根目录下或其它盘的根目录。把光盘根目录下的 sjf2440_Rev02/giveio/giveio.sys 文件复制到 C:\winnt\system32\drivers\目录下。

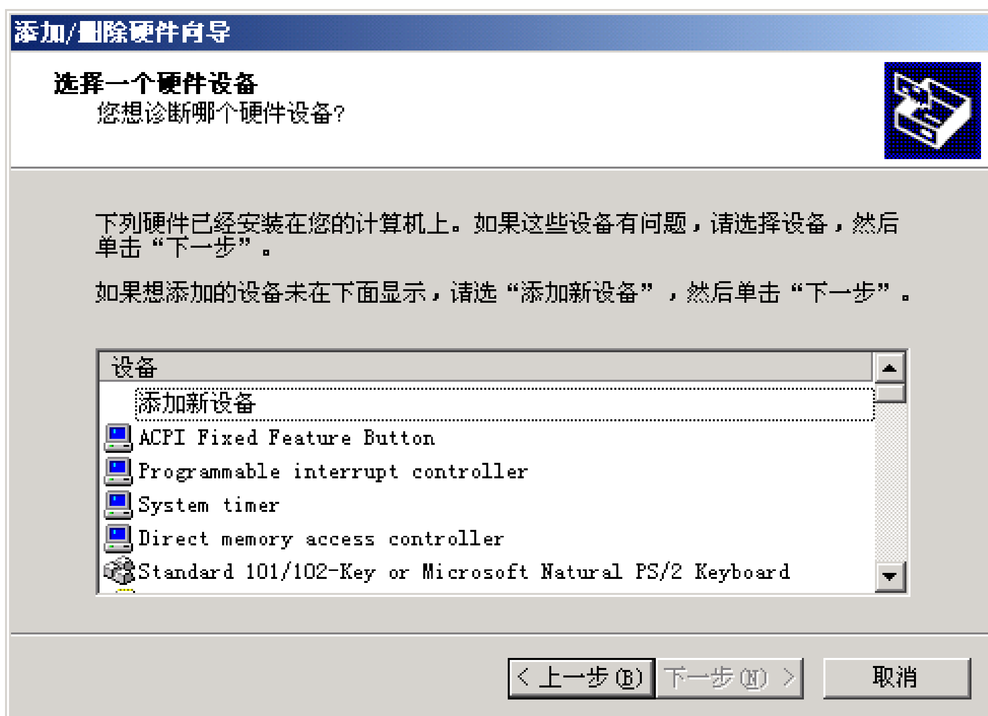
打开控制面板，选择添加删除硬件



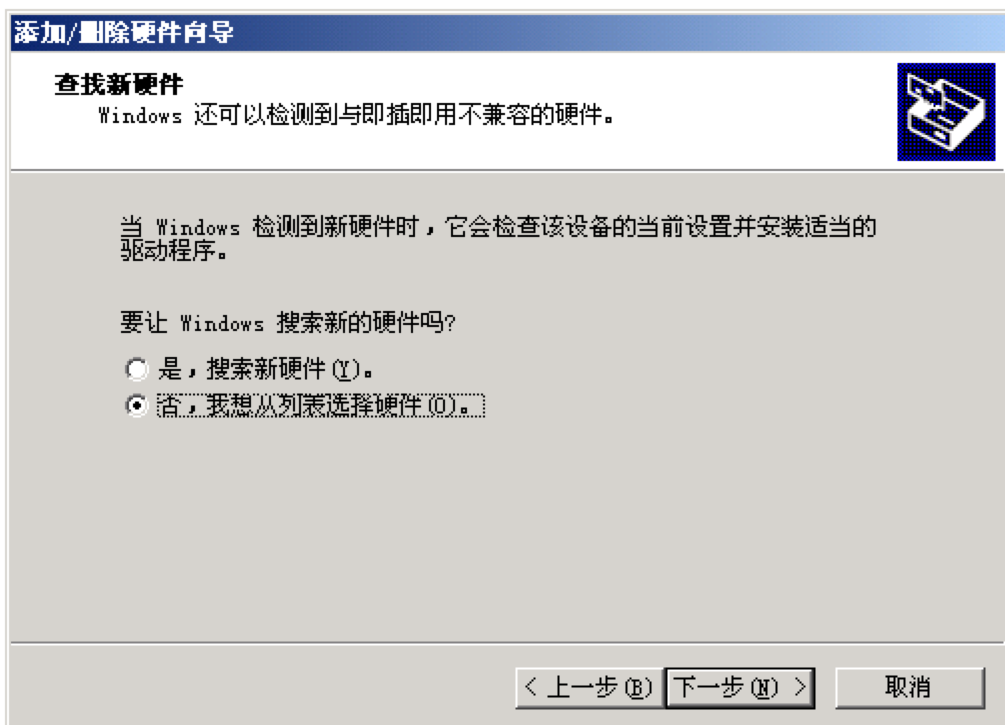
下一步，选择添加/排除设备故障



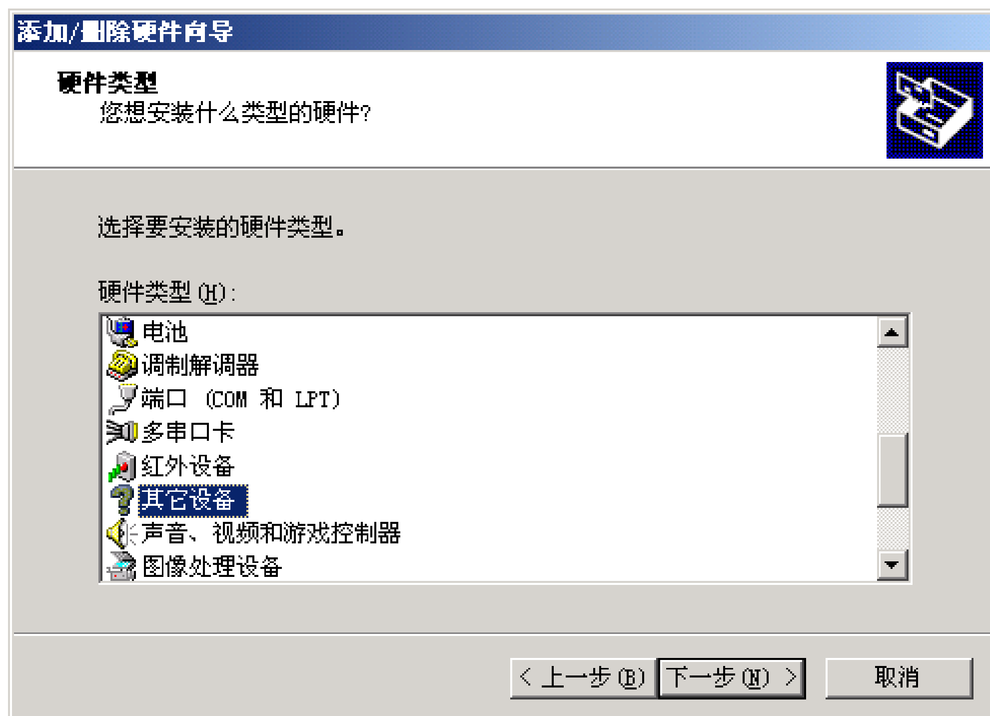
4．选择“ 添加新设备 ”，下一步



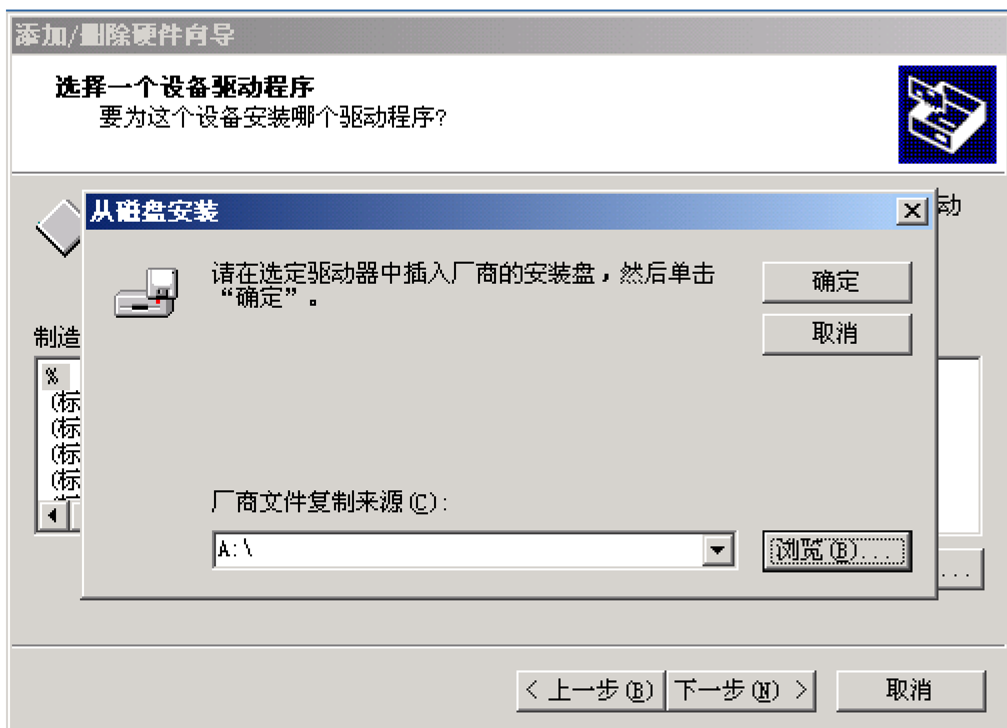
5. 选择“否，我想从列表选择硬件”，下一步



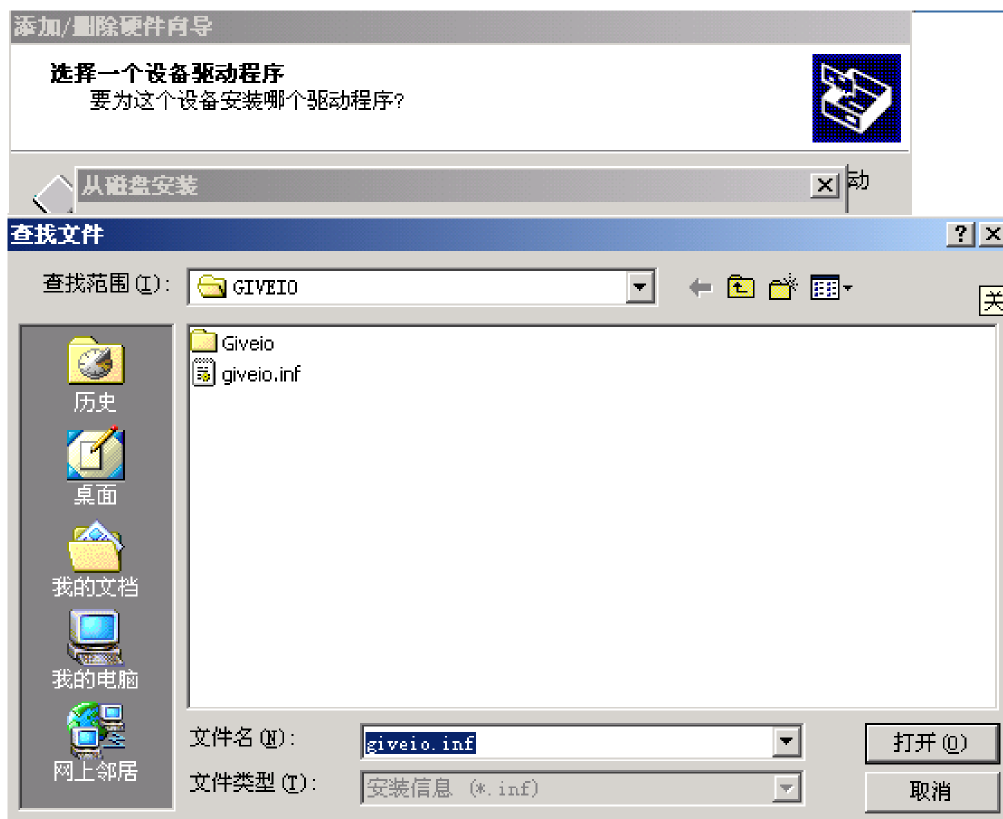
6．选择“其它设备”，下一步



7．选择“从磁盘安装”



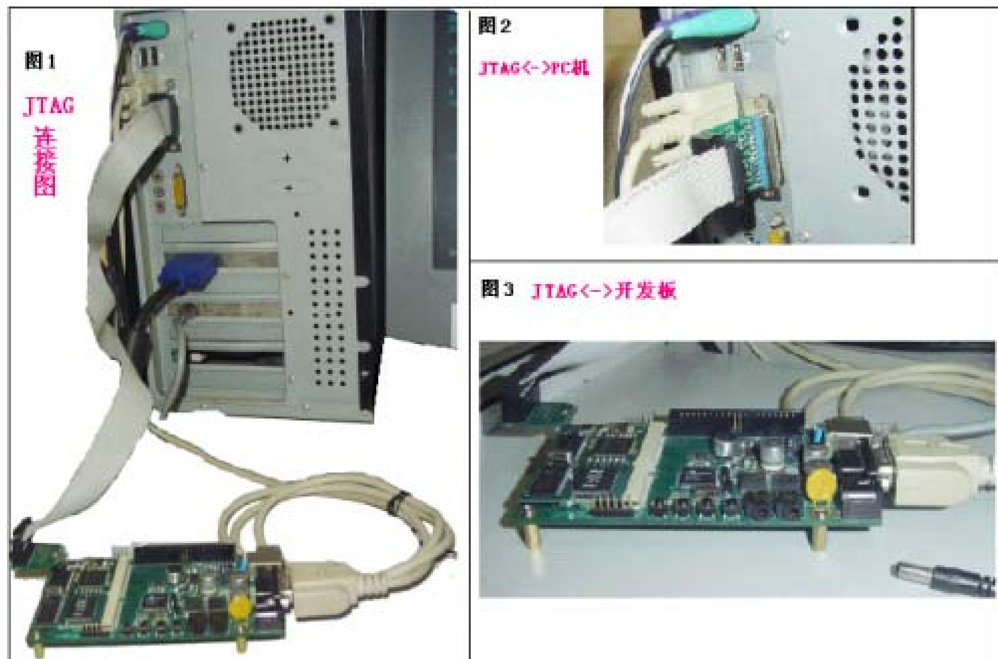
8. 选择“浏览”，选择 D:\sjf2440_Rev02\GIVEIO\giveio.inf，单击“打开”。



9．确定，完成

开始烧写 u-boot

硬件连接：把 JTAG 的一头接到 PC 机的并口，另一头接到 HHARM2440 的核心板的正面右上方,连接图如下：



最后加上电源。

jtag 线有凸起的那边朝向核心板的外面插到核心板的 jtag 插针上。如果使用了 20 针->10 针的转接头，当插上 JTAG 到核心板上，留出 PCB 板多的一方朝向核心板的外面。如果插反了，会提示您没有找到 CPU，**请不要带电拔插 JTAG。**

JTAG 工具完成板卡硬件检测、下载、烧写 FLASH 等最底层的调测功能。

在/HHARM2440/images/目录已经包含了二进制的引导代码 u-boot.bin 文件，内核映像文件 zImage 和文件系统映像文件 ramdisk.image.gz，这些文件一般情况上下就是开发板出厂时烧写的几个文件，如果客户因烧写了自己制作的文件出现问题，这几个文件可以拿来直接使用。

如果对 u-boot 做了某些改动，例如，修改了 TFTP 服务器的 IP 地址等等，则需要重新编译 u-boot，编译的步骤如下：

```
cd  /HHARM2440/u-boot-1.1.1
make distclean
make smdk2440_config
make
```

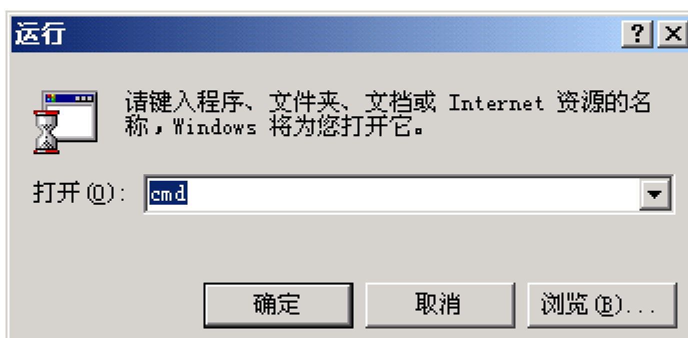
或运行 u-boot-1.1.1 目录下的 HHTECH 字母打头的文件即可生成新的 u-boot.bin 文件。通过 ls –l /HHARM2440/u-boot-1.1.1/u-boot.bin，确认 u-boot.bin 是否为新生成的文件。

1.把 JTAG 线插到 windows 2000 PC 机的并口，另一头接到 HHARM2440 开发板核心板的 10 针插头，给核心板加电。

『说明』:JTAG 插在核心板的正确接法是留出 PCB 板多的一方朝向核心板的外面。**禁止带电拔插 JTAG。**

2. 把 sjf2440.exe 和 u-boot.bin 复制到 D 盘的根目录下,省去进入很深的目录。

3. 打开 windows 2000 “开始菜单”，选择“运行”命令，输入 cmd,进入 windows 命令行，键入“D：”，进入 D 盘目录下




```
C:\WINNT\system32\cmd.exe
C:\Documents and Settings\administrator>d:

D:\>sjf2440 /f:u-boot.bin

+------------------------------------+
|     SEC JTAG FLASH(SJF) v 0.1      |
|     (S3C2440X & SMDK2440 B/D)      |
+------------------------------------+
Usage: SJF /f:<filename> /d=<delay>
> S3C2440X(ID=0x0032409d) is detected.

[SJF Main Menu]
 0:K9S1208 prog     1:28F128J3A prog     2:AM29LV800 Prog     3:Memory Rd/Wr
 4:Exit
Select the function to test:1

[28F128J3A Flash JTAG Programmer]

*** Very Important Notes ***
1. 28F128J3A must be located at 0x0(Bank0) for programming.
2. After programming, 28F128J3A may be located at 0x0(Bank0) for booting.

Source size = 15724h

Available Target Offset Address:
0x0,0x20000,0x40000, ...., 0x1ce0000
Input target address offset [0x?] : 0
```

4. 因为我们已经把 sjf2440.exe 、u-boot.bin 文件已经复制到 D 盘根目录了，键入以下命令开始把 u-boot.bin 文件烧写到 flash 中。

sjf2440.exe /f:u-boot.bin

出现提示信息选择 flash 类型：

Select the function to test:

键入 1 选择: 28F128J3A prog

出现提示信息选择确实烧写到 flash 中的位置：

1. Input target address offset [0x?] :

键入 0 选择：0 就会看见以下烧写提示信息。

```
选定 C:\WINNT\system32\cmd.exe
Source size = 15724h

Available Target Offset Address:
0x0,0x200000,0x400000, ...., 0x1ce0000
Input target address offset [0x?] : 0
Target base address(0x0) = 0x0
Target offset       (0x0)          = 0x0
Target size         (0x200000*n)   = 0x15724

Erase the sector from 0x0.
Block 00h Erase O.K.
Block 010000h Erase O.K.

Blank check is skipped.

Start of the data writing...
[100][200][300][400][500][600][700][800][900][a00][b00][c00][d00][e00][f00][1000
][1100][1200][1300][1400][1500][1600][1700][1800][1900][1a00][1b00][1c00][1d00][
1e00][1f00][2000][2100][2200][2300][2400][2500][2600][2700][2800][2900][2a00][2b
00][2c00][2d00][2e00][2f00][3000][3100][3200][3300][3400][3500][3600][3700][3800
][3900][3a00][3b00][3c00][3d00][3e00][3f00][4000][4100][4200][4300][4400][4500][
4600][4700][4800][4900][4a00][4b00][4c00][4d00][4e00][4f00][5000][5100][5200][53
00][5400][5500][5600][5700][5800][5900][5a00][5b00][5c00][5d00][5e00][5f00][6000
][6100][6200][6300][6400][6500][6600][6700][6800][6900][6a00][6b00][6c00][6d00][
6e00][6f00][7000][7100][7200][7300][7400][7500][7600][7700][7800][7900][7a00][7b
00][7c00][7d00][7e00][7f00][8000][8100][8200][8300][8400][8500][8600][8700][8800
][8900][8a00][8b00][8c00][8d00][8e00][8f00][9000][9100][9200][9300][9400][9500][
9600][9700][9800][9900][9a00][9b00][9c00][9d00][9e00][9f00][a000][a100][a200][a3
```

```
选定 C:\WINNT\system32\cmd.exe
4600][4700][4800][4900][4a00][4b00][4c00][4d00][4e00][4f00][5000][5100][5200][53
00][5400][5500][5600][5700][5800][5900][5a00][5b00][5c00][5d00][5e00][5f00][6000
][6100][6200][6300][6400][6500][6600][6700][6800][6900][6a00][6b00][6c00][6d00][
6e00][6f00][7000][7100][7200][7300][7400][7500][7600][7700][7800][7900][7a00][7b
00][7c00][7d00][7e00][7f00][8000][8100][8200][8300][8400][8500][8600][8700][8800
][8900][8a00][8b00][8c00][8d00][8e00][8f00][9000][9100][9200][9300][9400][9500][
9600][9700][9800][9900][9a00][9b00][9c00][9d00][9e00][9f00][a000][a100][a200][a3
00][a400][a500][a600][a700][a800][a900][aa00][ab00][ac00][ad00][ae00][af00][b000
][b100][b200][b300][b400][b500][b600][b700][b800][b900][ba00][bb00][bc00][bd00][
be00][bf00][c000][c100][c200][c300][c400][c500][c600][c700][c800][c900][ca00][cb
00][cc00][cd00][ce00][cf00][d000][d100][d200][d300][d400][d500][d600][d700][d800
][d900][da00][db00][dc00][dd00][de00][df00][e000][e100][e200][e300][e400][e500][
e600][e700][e800][e900][ea00][eb00][ec00][ed00][ee00][ef00][f000][f100][f200][f3
00][f400][f500][f600][f700][f800][f900][fa00][fb00][fc00][fd00][fe00][ff00][1000
0][10100][10200][10300][10400][10500][10600][10700][10800][10900][10a00][10b00][
10c00][10d00][10e00][10f00][11000][11100][11200][11300][11400][11500][11600][117
00][11800][11900][11a00][11b00][11c00][11d00][11e00][11f00][12000][12100][12200]
[12300][12400][12500][12600][12700][12800][12900][12a00][12b00][12c00][12d00][12
e00][12f00][13000][13100][13200][13300][13400][13500][13600][13700][13800][13900
][13a00][13b00][13c00][13d00][13e00][13f00][14000][14100][14200][14300][14400][1
4500][14600][14700][14800][14900][14a00][14b00][14c00][14d00][14e00][14f00][1500
0][15100][15200][15300][15400][15500][15600][15700]
End of the data writing

verifying is skipped.

D:\>
```

则正常，尤其是其中这一句> S3C2440X(ID=0x0032409d) is detected.可以判断 jtag 是否已经加上，若此句变成 ERROR: No CPU is detected(ID=0xffffffff).则有可能是 jtag 没有插好，或是插反了，或是 jtag 坏了。

烧写需要花上约 10 分钟的时间，请耐心等待。如果您已经烧写了 u-boot 而需要更改为新的 u-boot 时，可以在 u-boot 提示符下，通过 tftp 下载新的 u-boot，然后烧写来更新自身。当然也可以使用 JTAG 来烧写，但需要在 u-boot 提示符下键入解开第一个扇区的保护才能通过 JTAG 烧写。烧写成功以后，后面的工作就由 u-boot 接管。

3.2 软件应用开发

3.2.1 开发模式

在进行开发前，有必要先阐述一下宿主机和目标板的概念。宿主机是一台运行 LINUX 的 PC 机，目标板即华恒 HHARM2440 开发板。应用程序的开发有两种模式：

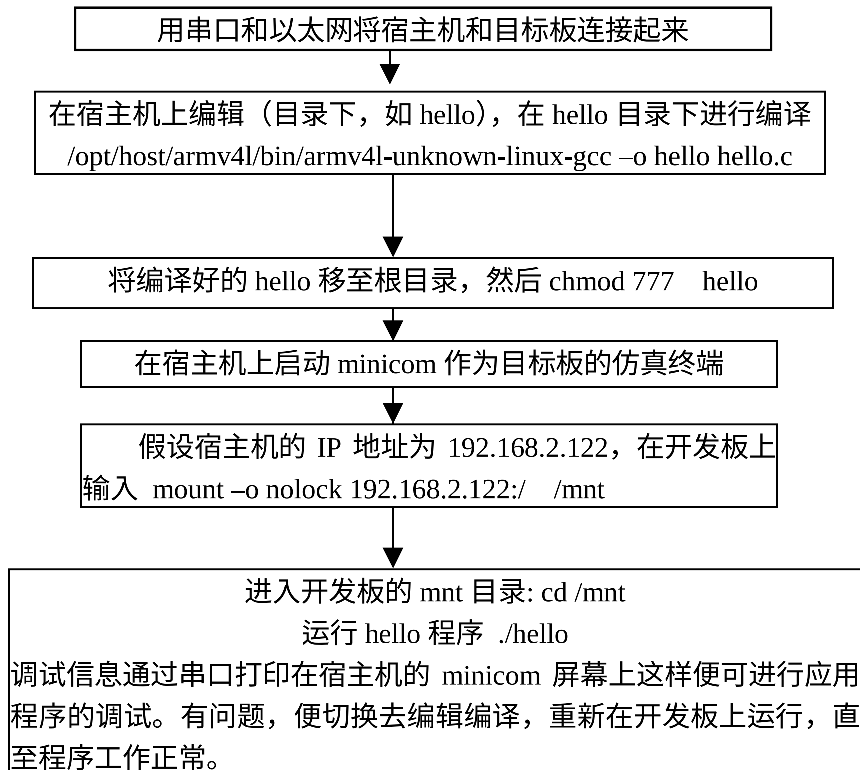
1. 先在宿主机（Intel CPU）上调试通过后，再移植到目标板（HHARM2440）上。移植的工作包括两个方面：

A．函数库的问题。在程序移植时可能会有函数未定义的问题。对于这种问题，一

般要求开发者自己编制这些要用到却又未定义的函数。

B．修改 Makefile 以选择适合目标板的编译工具

2. 直接在目标板上进行开发（**通用开发模式，建议采用该模式**）。将宿主机和目标板通过以太网连接，在宿主 PC 机上运行 minicom 作为目标板的显示终端，在目标板上通过 NFS (网络文件系统) 来 mount 宿主机硬盘，让应用程序直接运行在目标板上进行调试。下面给出这种直接 TARGET 开发模式下的开发流程：



3.U 盘调试法：(需要有 USB host 支持)

取消 U 盘的写保护，连接到宿主机上，执行

```
mount  /dev/sda1  /mnt
cd  /mnt
cp   /HHARM2440/applications/hello/hello   ./
```

/* 假 设 /HHARM2440/application/hello/hello 是我们要调试的应用 */

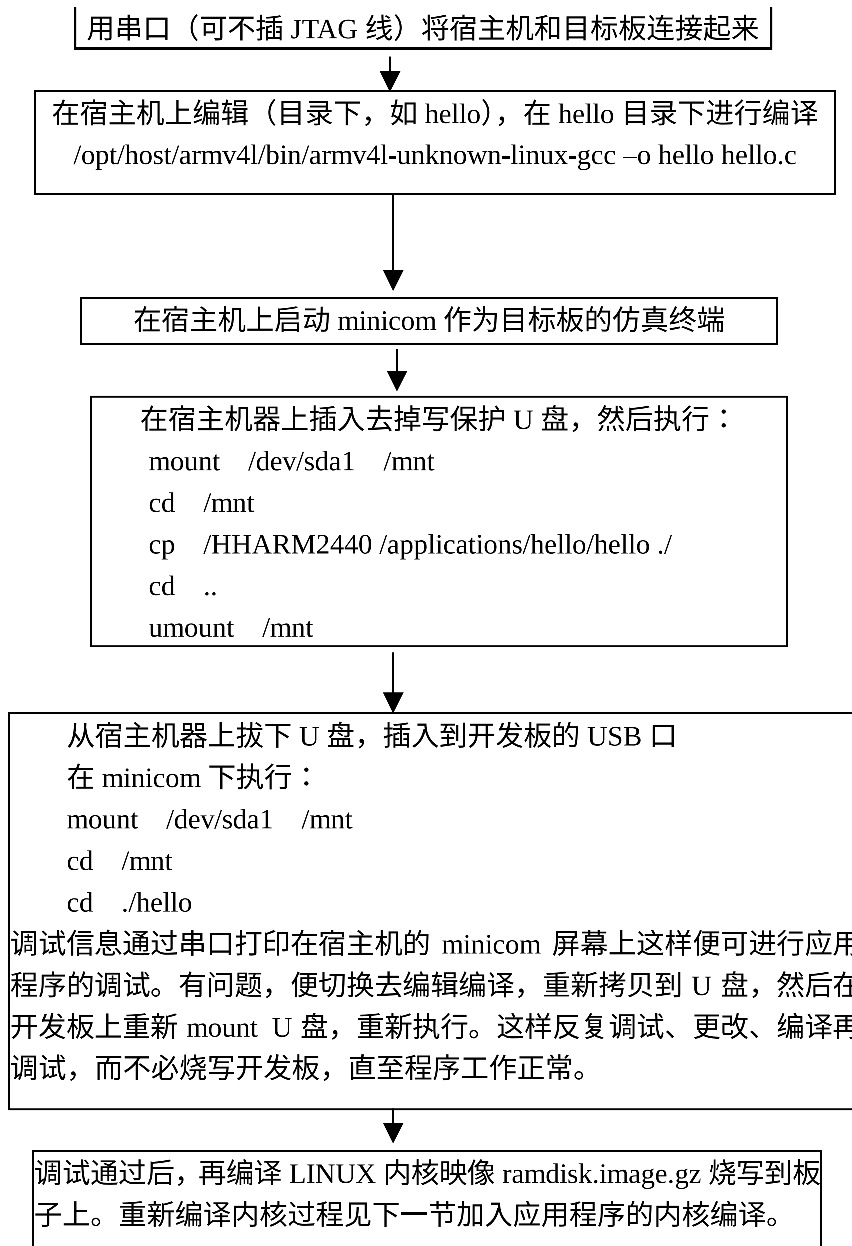
```
cd  ..
umount  /mnt
```

取下 U 盘，插入开发板的 USB 接口，执行

```
mount  /dev/sda1  /mnt
cd  /mnt
./hello
```

就可以看到程序在开发板上运行的情况。调试成功以后，把应用程序添加到 ramdisk 文件系统映像中，请参阅 3.2.3,制作好以后下载，烧写新的 ramdisk 映像文件。

下面给出这种通过 U 盘直接 TARGET 开发模式下的开发流程：



3.2.2 如何创建编译自己的应用

代码编写前应多阅读一下类似的应用程序的代码或从网上查找相关代码下载后阅读。

函数调用可参见《UNIX 环境高级编程》（见附录 B），因为书中所述为 UNIX 环境开发，但基本类似，具体应用到 Linux 时请使用 man，info 查看帮助。在程序移植时可能会有函数未定义的问题。对于这种问题，一般要求开发者自己编制这些要用到却又未定义的函数。

LINUX 下的应用程序全部都是用 C 代码开发的。用 C 代码开发应用程序，首先遇到的问题就是 C 库的问题，对于 HHARM2440 这种有 MMU 的处理器平台上的 LINUX 而言，就和 REDHAT LINUX 这种 PC LINUX 完全一样，用的都是 glibc，只不过是用不同的编译器编译而已。

HHARM2440 提供了编译好的 glibc 库的动态库（.so 文件）和静态库（.a 文件），安装在/opt/host/armv4l/armv4l-unknown-linux/lib 目录下，在应用程序中指定了编译器的路径后，就会自动链接这些库文件。

Makefile 可以参考/HHARM2440/application/下某个目录中的 Makefile。

下面举例说明：

在/目录下创建 hello 目录,编辑一个 hello.c 文件

```
cd  /
mkdir  hello
vi  hello.c
```

写入如下内容：

```
#include <stdio.h>
int main()
{
	printf("Hello World!\n");
	return 0;
}
```

编写 hello.c 只是打印串口 Hello world!字符串。编写 Makefile 文件编译器要用/opt/host/armv4l/bin/armv4l-unknown-Linux-gcc。以下是它的 Makefile 文件：

```
CC	= /opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CFLAGS = -DDEBUG -D__Linux__ -g
all:hello
hello: hello.o
		$(CC) $(CFLAGS) -o $@ $<
clean:
```

rm -rf *.o hello

然后执行 make，在目录 hello 目录下生成可执行文件 hello，最后将生成的可执行文件加入 Linux 文件系统中去，重新制作 ramdisk 文件系统映像并烧写 FLASH。重新制作 ramdisk 文件系统映像的过程见下一节加入应用程序的 ramdisk 文件系统映像制作。

3.2.3 加入应用程序的 ramdisk 文件系统映像制作

将生成的可执行文件加入 Linux 文件系统中去，重新制作 ramdisk 文件系统映像并烧写 FLASH。/HHARM2440/images/ramdisk.image.gz 为 LINUX 的文件系统映像压缩文件。用户可以在文件系统中加入自己的应用，例如可以将 ramdisk.image.gz 拷贝到根目录下并解开 ramdisk.image.gz：

```
cp  ramdisk.image.gz  /
cd  /
gunzip  /ramdisk.image.gz
```

此时根目录下会生成 ramdisk.image。ramdisk.image 为解压缩后的 Linux 的文件系统映像文件；再将 ramdisk.image 文件系统映像文件 mount 到/mnt 中。

```
mount  –o  loop  ramdisk.image  mnt
```

此时用户可以加入自己的应用程序 hello；

```
cd  /mnt
mkdir  app            //(新建目录名可以自己定义)
cd  app
cp  /hello  .
cd  /
umount  /mnt
```

现在压缩新生成的 ramdisk.image 文件系统映像文件：

```
gzip  ramdisk.image
cp  ramdisk.image.gz  /tftpboot/
```

下载烧写新的 ramdisk.image.gz 完毕后重启目标板，可以看到文件系统中出现了 app 目录，在 app 目录中出现了可执行文件 hello。。

```
# ls
app    bin    etc    hello    Linuxrc    mnt    sbin    usr
dev    lib    lost+found    proc    tmp    var
```

文件系统中出现了 app 目录

```
cd  app
```

在 app 目录中出现了可执行文件 hello,运行文件:

./hello

3.2.4 miniGUI 和 microwin 的移植

HHARM2440 套件提供 miniGUI 作为默认的 GUI，也提供 microwindows。它们的源代码目录在/HHARM2440/applications/目录下。

minigui移植

1.编译 minigui 的步骤大体如下：

A. 编译 minigui 库　　　　　　(在 libminigui-1.3.3 目录下操作)

B. 编译 minigui 支持资源　　　(在 minigui-res-1.3.3 目录下操作)

C. 编译 minigui 的示例程序　　(在 mg-samples-1.3.1 和 mde-1.3.0 目录下操作)

D.把以上步骤编译出来的库文件、资源文件、示例程序文件放到文件系统映像（ramdisk.image.gz）文件去

『说明』为了方便，我们在上述目录和 nfsroot 目录都放入一个 HHTECH 字母打头脚本文件，执行相应脚本文件，就可以完成一系列的操作，可以使用 vi 打开相关文件看看，所做的操作将在下面介绍移植过程作出介绍。

移植过程：

1.下载 miniGUI 软件包

从 http://www.minigui.com/download/cindex.shtml 可以下载到 minigui 的源代码及相关手册：

```
libminigui-1.3.3.tar.gz      //minigui 的库文件源代码
minigui-res-1.3.3.tar.gz     //minigui 的图标，字体等资源文件
mg-samples-1.3.0.tar.gz      //minigui 的简单示例
mde-1.3.0.tar.gz             //minigui 的综合示例
```

2.建立 miniGUI 开发目录

我们在/HHARM2440/applications 目录下建立 minigui-free 目录,将上述几个源代码包解压缩到此目录下，生成了 libminigui-1.3.3、minigui-res-1.3.3、mde-1.3.0、mg-samples-1.3.1。另外再建立 nfsroot 目录，用以存放生成的库文件、资源、示例等。等编译完上述源代码后，接下来要做的工作就是把 nfsroot 中的相关文件放到 ramdisk 文件系统中去。

3.配置 miniGUI 的 lib 库

为了编译出华恒 HHARM2440 平台的 minigui，我们修改了 libminigui-1.3.3 目录下的 configure 文件中前几行的 GNU 编译器，指定使用 ARM 的编译器进行编译，否则编译出来的库文件是 X86 平台的。

```
########HHTECH########################################################
CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-cpp
LD=/opt/host/armv4l/bin/armv4l-unknown-linux-ld
AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar
RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib
STRIP=/opt/host/armv4l/bin/armv4l-unknown-linux-strip
###############HHTECH##################################################
```

修改完以后，保存、退出，然后执行以下命令就可以编译出 HHARM2440 平台上支持 minigui 的库文件，并且把库文件拷贝到 nfsroot/lib 目录下。

```
./configure --host=arm-unknown-linux --enable-jpgsupport=no
--enable-pngsupport=no --enable-gifsupport=no --disable-lite
--prefix=/HHARM2440/applications/minigui-free/nfsroot
-enable-smdk2410ial=yes
make
make install
```

『说明』**以上命令我们已经写成了 HHTECH.inst.lib 脚本文件，直接执行此脚本文件相当于执行上述命令，同样地，后面的资源和示例源代码目录，我们也放置了一个这样的脚本文件。**

4.修改 libminigui-1.3.3 库，让 miniGUI 与 linux 内核通信

对 lib 库中源文件的修改仅限于 libminigui-1.3.3src/ial2410.c 文件，具体细节请参见此文件中含有 HHTECH 标记的部分，目的是告诉 minigui 确定触摸屏的设备名和如何从内核中读取触摸屏的坐标值。

5. 编译配置两种示例程序

和修改 lib 库源代码目录下的 configure 中的 GNU 编译器一样，我们修改了 mg-samples-1.3.1 和 mde-1.3.0 目录下的 configure 文件，在 configure 文件中加入以下几行指定使用 ARM 的编译器：

```
########HHTECH########################################################
CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-cpp
LD=/opt/host/armv4l/bin/armv4l-unknown-linux-ld
AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar
RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib
STRIP=/opt/host/armv4l/bin/armv4l-unknown-linux-strip
###############HHTECH##################################################
```

执行以下命令编译 mg-samples-1.3.1 或 mde-1.3.0。

```
./configure --build=i686-pc-linux-gnu --host=arm-unknown-linux
```

```
--prefix=/HHARM2440/applications/minigui-free/nfsroot/
LDFLAGS=-L/HHARM2440/applications/minigui-free/nfsroot/lib
CPPFLAGS=-I/HHARM2440/applications/minigui-free/nfsroot/include
CFLAGS=-I/HHARM2440/applications/minigui-free/nfsroot/include
Make
```

由于我们编译 minigui 时采用的是非 lite 模式，综合示例中的 mginit 没有办法演示,但其它像 bomb,fontdemo 等是可以运行的。HHARM2440 平台中我们没有存放综合示例，所以，mde-1.3.0 也可以不编译。

6.配置 miniGUI 示例支持资源

在 minigui-res-1.3.3 目录中修改 config.linux 文件,配置 miniGUI 执行示例时需要的资源，像字体，图标等等，指明 TOPDIR=/HHARM2440/applications/minigui-free/nfsroot。然后执行以下命令：

```
make install
```

就可以把相关文件拷贝到/HHARM2440/applications/minigui-free/nfsroot/目录下的相关目录中去。

在我们提供的/HHARM2440/applications/minigui-free/nfsroot.OK 目录中是已经做好的相关文件备份，其中删除了一些多余的字体、图标等，因为 ramdisk 文件系统放不下那么多的资源。

7. 裁剪生成的 lib 库

经过上述编译步骤，已经把 minigui 运行时所需要的运行环境的文件全部放在 nfsroot 目录下的了，为了把 minigui 的库文件，资源文件，示例程序放到 ramdisk 文件系统中去，我们需要对库文件和资源文件进行裁剪，让 miniGUI 占用更少的空间。

```
cd  /HHARM2440/applications/minigui-free/nfsroot/
rm –rf lib/minigui
```

// lib/minigui 目录下的文件和编译资源文件时生成的 nfsroot/usr/etc/local/lib/minigui 目录下的文件重复了。

```
rm –f lib/*.a   lib/*.la
```

// 因为我们采用的动态连接，删除所有的静态链接库，删除*.a 和*.la 文件，

```
/opt/host/armv4l/bin/armv4l-unknown-linux-strip  lib/*
```

// 删除库文件中的符号和调试等信息

经过以上步骤以后,nfsroot/lib 目录下的库文件大约占 800KB 右右的空间。

8. 制作 ramdisk 文件系统映像文件（ramdisk.image.gz）

拷贝 nfsroot/lib 中所有的库文件到 ramdisk 的/lib 中，将 nfsroot/usr/local/lib/minigui 目录 copy 到 ramdisk 的/lib 目录 中。同时在 ramdisk 中的/usr/local 目录中建立链向 ramdisk 中的/lib 目录的连接 ln –s

/lib lib。

拷贝 nfsroot/etc/MiniGUI.cfg 文件到 ramdiak 中的/etc(2004 年以前的开发板运行后，有一个拷贝动作，把/mnt/etc 拷贝到/etc，如果这样的 ramdisk.image.gz，请把 MiniGUI.cfg 拷贝到/mnt/etc)目录中去。可以参考一下我们 nfsroot.OK/etc 目录下的 MiniGUI.cfg 文件的做法。

拷贝 mg-samples-1.3.1/src/hello 或 treeview 等 miniGUI 的演示示例到 ramdisk 中的/bin 目录。

生成的资源文件也是挺大的，可以删除 nfsroot/usr/local/lib/minigui/res/font 目录下的一些字体文件和一些图标文件，删除时也要注意，MiniGUI.cfg 文件中指定的字体必须能找到，否则删除字体后，一些示例显示时会出现乱码，关于此，可以参照我们 ramdisk.image.gz 文件中的字体文件和 MiniGUI.cfg 文件的做法。

制作 ramdisk 文件系统的详细步骤如下：

```
cp /HHARM2440/images/ramdisk.image.gz   /
gunzip ramdisk.image.gz
mount －o loop ramdisk.image   /mnt
            //解开 ramdisk 文件系统压缩文件，并把其 mount 到 PC 机的/mnt
cd mnt
mv /HHARM2440/applications/minigui-free/nfsroot/lib/*  /mnt/lib
            //拷贝或移动，拷贝时，cp 要带一些参数，把连接文件也一道拷贝
cd  /mnt/lib
cp －rf /HHARM2440/…………/nfsroot/usr/local/lib/minigui  .
            //minigui 中的资源文件占的空间太大，一定要剪裁一下。
cp /HHARM2440/…………/nfsroot/etc/MiniGUI.cfg  /mnt/etc
            //这时要注意，此 MiniGUI.cfg 如果配置不正确，示例程序可能无
            法运行，请参考我们的 ramdisk.image.gz 中的此文件。
cp /HHARM2440/…………/mg-sample-1.3.1/src/treeview  /mnt/bin
cd  /mnt/usr/local
ln －s   /lib   lib
            //minigui 默认调用/usr/local/lib/minigui 目录下资源文件，上
            面已经把 minigui 目录放到/lib 目录下去了，所以，建立一个链接。
cd   /
umount /mnt
gzip ramdisk.image
cp ramdisk.image.gz   /tftpboot
```

现在，就可以把内核映像和压缩的文件系统映像下载到开发板的SDRAM中运行一下(别急着烧写)，如果一切顺序，在开发板启动以后，运行一下treeview就可

以看到minigui漂亮的界面了。

microwindows的移植

Microwindows 的源代码可以从 http://www.microwindows.org/ 下载，microwindows 的源代码以及相关文档在/HHARM2440/applications/microwindows 下面，源码在/HHARM2440/applications/microwindows/src 下面。

配置，编译 microwindows:

microwindows 提供了图形化的配置工具，在 X-windows 中，在源代码目录下可运行下面两个命令之一（两者是等价的）启动图形化配置。

```
./xconfigure
make xconfig
```

1.选择平台,单击 Platform 按钮后,我们选择 Linux-ARM。

在 Option 中有一些于平台相关的设定,Tools Prefix 设定的是交叉编译器的前缀,在这里是/opt/host/armv4l/bin/armv4l-unknown-linux-,

它表示编译时要用的交叉编译器是/opt/host/armv4l/bin/armv4l-unknown-linux-gcc, 链接器是/opt/host/armv4l/bin/armv4l-unknown-linux-ld 等。

如果需要支持 LCD,需要在 screen driver 中选上 framebuffer,如果需要支持触摸屏,需要在 mouse driver 中选上 touch pad。

2.选定要生成的库。在 Libraries to compile 中选上 Nanox,Microwindows,这两个库是很多 microwindows 应用程序都需要的库.编译后生成的库(包括其它库)都在./lib 下面（假设当前目录是 microwindows 源码目录)。

3.还有一些其他的选项,用户可以根据其中的自己的需要进行配置。

配置好了以后,make 以后就生成 microwindows 的应用程序，根据配置可能包括一些示列程序。

microwindows 和 Nano-X

microwindows 支持两种 API,一种是和 win32 兼容的 API,另一种是类似于 Xlib 的 API,也就是 Nano-X.　　后者用得更广泛一些，基于 Nano-x API 的应用程序是通过客户/服务器的方式运行，类似于 X-windows 程序。图像服务器程序是安装是生成的，即源码目录的 bin 目录下的 nano-X。在运行 nano-X 应用程序之前，需要启动 nano-X 服务器程序。

在编译 microwindow 后,bin 下面还有一些其他应用程序,如果是基于 nano-X 通过：

```
nano-X&;program paras
```

来运行，如 nxclock，其中 program 是可执行程序名，papas 是参数。如果不是 nano-x 应用程序，则直接通过：

program paras

运行,如 malpha。

一个 nano-X 示例程序

下面是一个简单的基于 nano-x 库的应用程序，向我们简单展示 nano-X 程序的编写，编译和运行。

```
//hello.c ---a nano-X program example
#include <stdio.h>
#include "nano-X.h"
GR_WINDOW_ID wid;
GR_GC_ID gc;
void event_handler (GR_EVENT *event);
int main (void)
{
   if (GrOpen() < 0) {
        fprintf (stderr, "GrOpen failed");
        exit (1);
   }
   gc = GrNewGC();
   GrSetGCForeground (gc, 0xFF0000);
   wid = GrNewWindowEx(GR_WM_PROPS_APPFRAME |
                       GR_WM_PROPS_CAPTION  |
                       GR_WM_PROPS_CLOSEBOX,
                       "jollen.org",
                    GR_ROOT_WINDOW_ID,
                       0, 0, 200, 200, 0xFFFFFF);
   GrSelectEvents(wid,          GR_EVENT_MASK_CLOSE_REQ            |
GR_EVENT_MASK_EXPOSURE|GR_EVENT_MASK_MOUSE_ENTER|GR_EVENT_MASK_MOUSE_E
XIT|GR_EVENT_MASK_MOUSE_MOTION|GR_EVENT_MASK_MOUSE_POSITION);
   GrMapWindow(wid);
   GrMainLoop(event_handler);
}

void event_handler (GR_EVENT *event)
{
   switch (event->type)
   {
      case GR_EVENT_TYPE_EXPOSURE:
```

```
            GrText(wid, gc, 50, 50, "Hello World", -1, GR_TFASCII);
        break;
       case GR_EVENT_TYPE_CLOSE_REQ:
        GrClose();
            exit (0);
       default: break;
    }
  }
```

GrOpen()函数建立与 nano-X 服务器的连接,GrNewGc()建立一个图形上下文环境，GrNewWindowEx()建立一个窗口，最后通过 GrMainLoop()事件循环，这些函数都是 nano-X 的库函数，可参阅 Nano-X Programming Tutorial （http://www.microwindows.org/Nano-XTutorial.html）， nano-X.h 是应用程序需要包含的头文件。

Makefile 文件为：

```
##Makefile for hello application
CC = /opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CFLAGS = -Wall -I/HHARM2440/applications/microwindows/src /include
-I/HHARM2440/applications/microwindows/src/drivers
-L/HHARM2440/applications/microwindows/src/lib
LIBS = -lmwin -lmwinlib -lmwengine -lmwdrivers -lmwfonts -lnano-X
OBJECTS = hello.o
all: hello
hello: $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) $(LIBS) -o hello
clean:
    rm -f *.o hello
```

在这里面，需要添加 nano-X 头文件和库函数的搜索路径，分别用-I 和-L 指定。并指定需要相关库，如 libnono-X,libmwin。

编译后，可通过运行

nano-X &

./hello (假设可执行程序 hello 已经在当前目录)

来运行该程序，该程序是显示一个打印 “ hello world ” 窗口。

在 Linux 这个开放的世界里，从网上下载相关软件并移植到给定的平台上将是一种常见的工作模式，其实就具体的工作而言，同本节介绍的内容没什么区别。

第四章 外围接口使用介绍

为了测试方便,我们在 ramdisk 文件系统的根目录下建立了 hhtech 这样的目录,并在其下建立相关子目录，执行/hhtech/test-script 目录下的相关脚本文件，就可以对外围接口进行测试。

4.1 摄像头

- 驱动程序文件位置：/HHARM2440/modules.TestApp/camera/drv
- 测试程序文件位置：/HHARM2440/modules.TestApp/camera/app

摄像头的驱动是以模块的方式加载的，编译以上的驱动程序及应用程序，键入以下命令对摄像头进行操作

insmod 2440camera.o

cam2fb 240 320 16 0

在 LCD 上就可以看到摄像头传过来的图像了。

4.2 USB 接口

- 驱动程序文件位置：/HHARM2440/linux-2.4.20/drivers/usb/

HHARM2440 接口底板提供了两个 USB HOST 接口和一个 USB Device 接口（其实其中一个 USB HOST 和一个 USB Device 接口复用，只能使用其中之一，即系统要么使用两个 USB HOST 接口，要么使用一个 USB HOST 和 USB Device），可支持 U 盘、USB 摄像头等多种 USB 设备,但使用不同的设备需要开发相应的设备驱动。华恒标准套件已经提供了 U 盘的驱动程序，用户可直接使用 U 盘,其它的驱动需要客户自已移植相关驱动。

要访问 U 盘用户需要把它 mount 上，当插入 U 盘时，会出现如下提示：

```
~ # hub.c: USB new device connect on bus1/2, assigned device number 2
Manufacturer: USB
Product: Solid state disk
SerialNumber: 3A5E15F23FAA1BD6
scsi0 : SCSI emulation for USB Mass Storage devices
  Vendor: KPM       Model: KPM              Rev: 1.11
  Type:   Direct-Access                     ANSI SCSI revision: 02
```

Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 32256 512-byte hdwr sectors (17 MB)
sda: Write Protect is on
Partition check:
 /dev/scsi/host0/bus0/target0/lun0: p1 //U 盘分区
即在 minicom 下执行如下命令：
~ # **mount /dev/scsi/host0/bus0/target0/lun0/part1 /tmp/** //挂载 U 盘
mount: /dev/scsi/host0/bus0/target0/lun0/part1 is write-protected, mounting reay
VFS: Can't find ext3 filesystem on dev sd(8,1).
mount: /dev/scsi/host0/bus0/target0/lun0/part1 is write-protected, mounting reay
VFS: Can't find ext2 filesystem on dev sd(8,1).
mount: /dev/scsi/host0/bus0/target0/lun0/part1 is write-protected, mounting reay
cramfs: wrong magic
mount: /dev/scsi/host0/bus0/target0/lun0/part1 is write-protected, mounting reay
~ # **cd /tmp/**
/etc/tmp # **ls** //列出 U 盘内容
1.mp3 ffplay rc-mp3 rc-recorder2
1.rar mp3play rc-mtv rc-video
2.rar mplayer rc-recorder v120x160.avi
『说明』：为了方便，可以在启动脚本文件中加入这样几行
mknod /dev/sda1 b 8 1mknod /dev/sda1 b 8 1
mknod /dev/sda2 b 8 2
mknod /dev/sdb1 b 8 17
mknod /dev/sdb2 b 8 18
mknod /dev/sdc1 b 8 33
这样 mount 以后，就可以用如下命令挂载 U 盘了。

mount –t ext2 /dev/sda1 /mnt

支持多种文件的 U 盘，如：EXT2，VFAT 等。
内核编译选项为：

```
cd kernel
make menuconfig
USB support  --->
<*> Support for USB
[*]   USB verbose debug messages
--- Miscellaneous USB options
[*]   Preliminary USB device filesystem
[ ]   Enforce USB bandwidth allocation (EXPERIMENTAL)
```

```
[ ]   Long timeout for slow-responding devices (some MGE Ellipse UPSes)
--- USB Controllers
< >   UHCI (Intel PIIX4, VIA, ...) support
< >   UHCI Alternate Driver (JE) support
< >   TransDimension UHC124 Driver support
< >   OHCI (Compaq, iMacs, OPTi, SiS, ALi, ...) support
< >   SA1111 OHCI-compatible host interface support
<*>   S3C2440 OHCI-compatible host interface support
(1)     Maximum port(s) of RootHub
--- USB Device Class drivers
< >   USB Bluetooth support (EXPERIMENTAL)
<*>   USB Mass Storage support
[ ]     USB Mass Storage verbose debug
[ ]     Datafab MDCFE-B Compact Flash Reader support
[ ]     Freecom USB/ATAPI Bridge support
```

4.3 LCD/触摸屏

- LCD 驱动程序主文件：...... /drivers/video/s3c2440fb.c
- 触摸屏驱动程序主文件：...... /drivers/char/s3c2440_ts.c

华恒 HHARM2440 标准套件中配置了 240×320 TFT 彩屏 LCD（带触摸屏）。可支持多种 LCD 屏，例如黑白单色屏、STN 彩屏已及 TFT 彩屏等。

此外还可定制支持各种其它的屏，例如 160×160 象素黑白单色触摸屏，10.4 寸 640×480 的大屏等。

套件提供的 Linux 操作系统为型号不同 LCD 的提供了相应的 framebuffer 底层驱动，可对 Microwindows，MiniGUI 等嵌入式图形系统提供良好的支持，而且都已经完成了对触摸屏的良好支持。

套件还提供了一套自定义的简单 GUI 库，即一套 GUI API 函数库，支持画点、画线、中英文显示及位图显示等，并应用这些函数库给出了一些 LCD 显示的例子程序，例如：gui，handpad，tp_gui 等，例程源代码在 /HHARM2440/applications/gui-demo 目录下。

LCD 的支持需要对内核进行配置：make menuconfig->Console drivers->Frame-buffer support->

Console drivers --->

Frame-buffer support --->

[*] Support for frame buffer devices (EXPERIMENTAL)

<*> S3C2440 LCD support
< > 96x320 emulation support
< > PXA LCD support
[ ] MediaQ Framebuffer support
< > ITE IT8181 framebuffer support
< > Virtual Frame Buffer support (ONLY FOR TESTING!)
[*] Advanced low level driver options
< > Monochrome support
< > 2 bpp packed pixels support
< > 4 bpp packed pixels support
< > 8 bpp packed pixels support
<*> 16 bpp packed pixels support
< > 24 bpp packed pixels support
< > 32 bpp packed pixels support
< > Amiga bitplanes support
< > Amiga bitplanes support
< > Amiga interleaved bitplanes suppor
< > Atari interleaved bitplanes (2 planes) support
< > Atari interleaved bitplanes (4 planes) support
< > Atari interleaved bitplanes (8 planes) support
< > Mac variable bpp packed pixels support
< > VGA 16-color planar support
< > VGA characters/attributes support
< > HGA monochrome support (EXPERIMENTAL)
[*] Support only 8 pixels wide fonts
[*] Select compiled-in fonts
[ ] VGA 8x8 font
[ ] VGA 8x16 font
[ ] Sparc console 8x16 font
[ ] Pearl (old m68k) console 8x8 font
[*] Acorn console 8x8 font

测试 LCD 及触摸屏：

启动开发板以后，执行以下命令就可以在 LCD 上看到 minigui 的示例程序了：

treeview

使用触摸笔就可以移动 LCD 中显示的 minigui 的示例程序 treeview 的窗口及展开和折叠目录树。

4.4 100M 以太网

- 驱动程序主文件：…… /drivers/net/dm9000x.c
- 测试程序文件位置：/HHARM2440/modules-TestApp/ethernet-performance-test

HHARM2440 通过外接一片 DM9000 以太网 MAC 芯片扩展了一个 10/100M 自适应的以太网接口，占用资源：nGCS1/EINT0。

内核编译选项为：

```
Network device support  --->
Ethernet (10 or 100Mbit)  --->
<*>      HHTECH_DM9000 ethernet controller
```

ethernet-performance-test 是测试以太网性能的程序。其中一个是运行在开发板上的，另一个是运行在 PC 端的

测试步骤：

开发板用网线（对接线）同 PC 机相连；

PC 机上编译好 eth-perf-pc/中的程序，生成 server 可执行程序，运行

./server

在 minicom 下执行客户端程序（开发板上）

./client 192.168.2.126 –t 1 –p 1

其中 192.168.2.126 为 PC 机的 IP,-t 表示 time，-p 表示 package。

4.5 IDE 硬盘

- 驱动程序主文件：…… /drivers/ide/cpci405ide.c

在 HHARM2440 上增加 IDE 接口，占用的硬件资源是片选 CS3、中断 EINT3、IDE 接口的地址空间的基地址是 0x18000000，16 位的 I/O 读写方式（寄存器读写时只用到低 8 位）。

硬件设计是通过可编程逻辑芯片直接把 IDE 接口的各信号线通过 buffer 连接到系统总线上， IDE 寄存器读写和 IDE 数据读写都采用 16 位的读写方式（由片选决定的，但对寄存器的读写只有高 8 位数据是有效的。

把 IDE 驱动加入到 linux 内核，在 make menuconfig 中选中“ ATA/IDE/MFM/RLL support ”，然后选择 “ IDE, ATA and ATAPI Block devices ” 下的如下选项：

(1) Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support (NEW)
(2) Include IDE/ATA-2 DISK support (NEW)
(3) Other IDE chipset support
(4) CPCI-405 IDE interface support

硬盘的使用方法：

把硬件连接好，启动开发板，

~ # **mount /dev/ide/host0/bus0/target0/lun0/** //按 Tab 键会出现以下提示

/dev/ide/host0/bus0/target0/lun0/disc

/dev/ide/host0/bus0/target0/lun0/part1

/dev/ide/host0/bus0/target0/lun0/part2

~ # **mount /dev/ide/host0/bus0/target0/lun0/part1 /tmp/** //把硬盘的第一个分区挂载到/tmp 目录下。

kjournald starting. Commit interval 5 seconds

EXT3 FS 2.4-0.9.17, 10 Jan 2002 on ide0(3,1), internal journal

EXT3-fs: recovery complete.

EXT3-fs: mounted filesystem with ordered data mode.

【注意】

上面打印的信息并不是表示出错，而是因为 /dev/ide/host0/bus0/target0/lun0/part1 分区是个 WINDOW2000 的系统，不是 ext2/ext3 文件系统格式。

~ # **cd /tmp/**

~ # **ls** //就可以看到硬盘中的内容了。

还可以进入以下目录，查看相关分区情况。

~ # **cd /proc**

~ # **cat partitions**

```
major minor  #blocks  name

   3     0   120060864 ide/host0/bus0/target0/lun0/disc
   3     1    60026841 ide/host0/bus0/target0/lun0/part1
   3     2           1 ide/host0/bus0/target0/lun0/part2
   3     5    60026841 ide/host0/bus0/target0/lun0/part5
```

~ # **cat devices**

Character devices:

1 mem

2 pty/m%d

3 pty/s%d

4 tts/%d

5 cua/%d

7 vcs

```
 10 misc
162 raw
200 ioport

Block devices:
  1 ramdisk
  3 ide0
  7 loop
 43 nbd
```

~ # **ls /dev/hda* -l**

```
brw-rw----    1 root     root       3,   0 Sep 12  2002 /dev/hda
brw-rw----    1 root     root       3,   1 Sep 12  2002 /dev/hda1
brw-rw----    1 root     root       3,  10 Sep 12  2002 /dev/hda10
brw-rw----    1 root     root       3,  11 Sep 12  2002 /dev/hda11
brw-rw----    1 root     root       3,  12 Sep 12  2002 /dev/hda12
brw-rw----    1 root     root       3,  13 Sep 12  2002 /dev/hda13
brw-rw----    1 root     root       3,  14 Sep 12  2002 /dev/hda14
brw-rw----    1 root     root       3,  15 Sep 12  2002 /dev/hda15
brw-rw----    1 root     root       3,  16 Sep 12  2002 /dev/hda16
brw-rw----    1 root     root       3,   2 Sep 12  2002 /dev/hda2
brw-rw----    1 root     root       3,   3 Sep 12  2002 /dev/hda3
brw-rw----    1 root     root       3,   4 Sep 12  2002 /dev/hda4
brw-rw----    1 root     root       3,   5 Sep 12  2002 /dev/hda5
brw-rw----    1 root     root       3,   6 Sep 12  2002 /dev/hda6
brw-rw----    1 root     root       3,   7 Sep 12  2002 /dev/hda7
brw-rw----    1 root     root       3,   8 Sep 12  2002 /dev/hda8
brw-rw----    1 root     root       3,   9 Sep 12  2002 /dev/hda9
```

【注意】

检测到的分区

```
   3     1   60026841 ide/host0/bus0/target0/lun0/part1
   3     2          1 ide/host0/bus0/target0/lun0/part2
   3     5   60026841 ide/host0/bus0/target0/lun0/part5
```

根据前面的 major 和 minor 号，分别对应 hda1/hda2/hda5。

~ # **mount /dev/hda1 /mnt**　　　　　//或者这样 **mount 硬盘。**

```
VFS: Can't find ext3 filesystem on dev ide0(3,1).
VFS: Can't find ext2 filesystem on dev ide0(3,1).
```

~ # **mount**
/dev/rd/0 on / type ext2 (rw)
/proc on /proc type proc (rw)
/dev/hda1 on /mnt type msdos (rw)
~ # **cd /mnt**
~ # **ls**

```
adobeweb.log  boot.ini      config.sys   mymusi~1      pagefile.sys
arcldr.exe    bootfont.bin  docume~1     ntddk         progra~1
arcsetup.exe  comreads.dbg  io.sys       ntdetect.com  recycled
autoexec.bat  comused.dbg   msdos.sys    ntldr         winnt
```

~ # **df** //查看各分区使用情况

```
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/rd/0                 3963      3960         0 100% /
/dev/hda1             60012160   4659904  55352256   8% /mnt
```

4.6 PS2 键盘

- 驱动程序主文件：…… /drivers/char/keyb_ps2.c
- 片选、中断：片选 nGCS4，中断 EINT4

目前可支持到接上 PS2 键盘可在 minicom 里面显示键值。

4.7 音频

- 驱动程序主文件：…… /drivers/sound/s3c2440-audio.c
- 录音程序文件位置：/HHARM2440/modules-TestApp/audio/app/
- 放音程序文件位置：/HHARM2440/applications/madplay、Mplayer-0.91

支持 ADPCM/PCM 录音和 WAV 文件播放。

因我们已经把相关测试脚本已经放到开发板中了，启动开发板，进入 /hhtech/test-script 目录下，执行相关脚本就可以录音和放音了，具体使用的。

1． 录音并播放。

recorder -n num //其中 num 为一个以秒为单位的数值。可以调整 “ num ” 的大小来改变录音时间。录音被保存在 “ /tmp/out.wav”中。

【注意】录音时间不能太长，否则内存空间不够。

播放刚才的的录音：

cp /tmp/out.wav /dev/dsp

或者 **cat /tmp/out.wav >/dev/dsp**

这样是播放开发板录的 wav 文件，如果 PC 机上拷贝到开发板上的 wav 文件，需要相应的播放软件才能控制驱动播放正常的声音，否则，有可能播放声音的速度快，有的慢。

2． 播放 MP3 音乐文件。

cd　/hhtech/test-script

./t-play-mp3

4.8 串口通信和 GPRS 拨号（可选）

- 驱动程序主文件：…… /drivers/char/serial.c
- 测试程序文件位置：/HHARM2440/modules-TestApp/serial-test/

HHARM2440 提供的第一个串口作为系统的标准输入输出，第二个串口可以用于通信。系统内已集成了完整的串口通信方案。

测试串口通信的步骤（ttytest）：

两个串口分别接 PC 机，并分别运行 minicom；在串口 1 所接的 minicom 窗口下键入 ttytest,回车；在串口 2 所接 minicom 窗口下看到收到的字符，键入字符，看到串口 1 所接 minicom 窗口下收到字符，说明串口 2 收发数据均正常；

此外，串口还有一个用途，就是接 MODEM 进行 PPP 拨号。

1、首先是根据开发板的串口 2 的信号定义制作串口线接 MODEM，要将接 MODEM 的串口线 2、3 交换，7、8 交换。做好线之后，可用 tip 软件进行与 MODEM 通信的测试：

tip -l /dev/ttyS1 -s 115200

显示 connected 后，用户就可以输入 AT 命令直接与 MODEM 对话了，这样做的目的是测试串口线。

2、配置内核支持 PPP，并编译拨号用相关应用程序：pppd/chat。关于内核支持，就是 make menuconfig 在内核配置的 Network device support ---> 中选择 PPP/SLIP/CSLIP 等，具体细节选项可任选，其实全部选中就可以了。关于应用程序的做法，和上面 boa 的做法完全类似。

3、编写拨号脚本和相关配置文件。做拨号必须先在 REDHAT 上手工拨号成功（不要用图像界面），然后才能到开发板上做测试。而且，这只是第一步，简单的拨号上去并没有任何意思，因为产品过程中，还要涉及许多的应用细节，例如断线重拨检测，开机自动拨号，无流量自动断开，有流量自动拨号等，都是非常麻烦的细节，需要做长时间大量的稳定性可靠性测试。

华恒提供经过全球各大城市测试稳定应用的拨号软件，可大大节省用户产品的上市时间。可联系华恒市场部购买 PPP(Modem/GPRS/CDMA 拨号，通过串口)拨号软件包和 PPPoe(ADSL 拨号，通过以太网口)拨号软件包，软件包中提供了 PPP 和

PPPoe 拔号的源代码和专用的说明文档。

4.9 实时时钟（RTC）

- 驱动程序主文件：...... /drivers/char/s3c2440-rtc.c
- 测试程序文件位置：/HHARM2440/modules-TestApp/RTC/app
  通过底板上备用电池供电。

1．设置时间命令格式为

rtc mmddhhmmssyyyy 即（rtc 月 日 小时 分钟 秒 年）。

月、日、小时、分钟、秒使用 2 位数字，不够前面补 0；年为 4 位数字。

2．读取 RTC 时钟的时间，直接在命令提示符下键入：

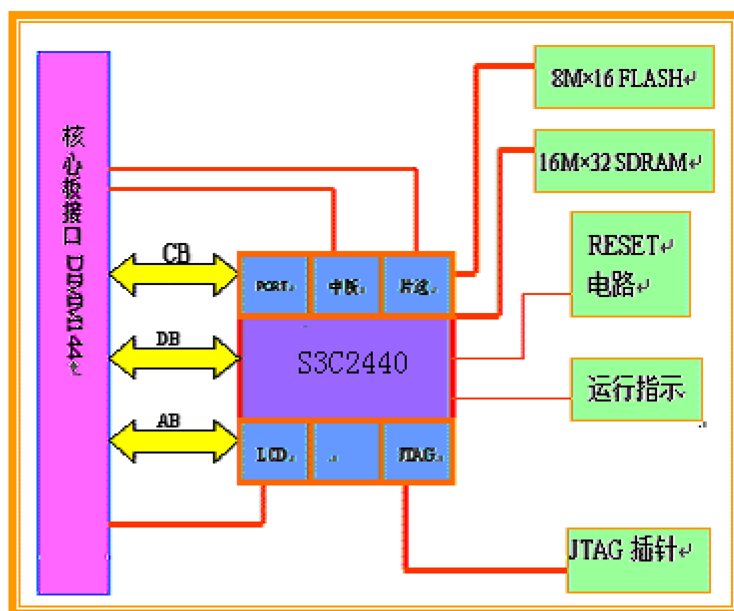
rtc

说明 ：以上的那个 rtc 测试程序的代码需要修改一才能读写正确的时间，或修改 busybox 源代码中的 date.c 文件，以像 PC 机的 date 一样，可以使用 date 命令来设置时间。

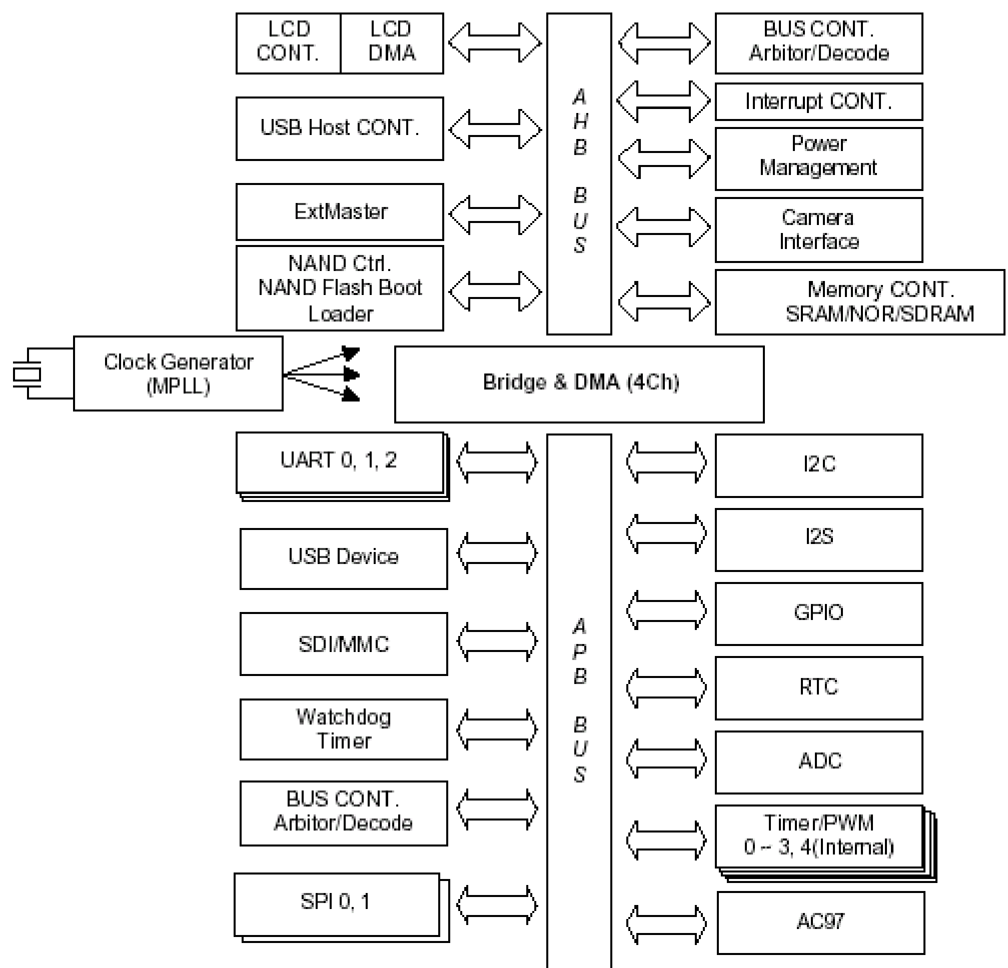
第五章 硬 件 系 统

5.1 功能模块结构图

核心板:



S3C2440 功能模块：



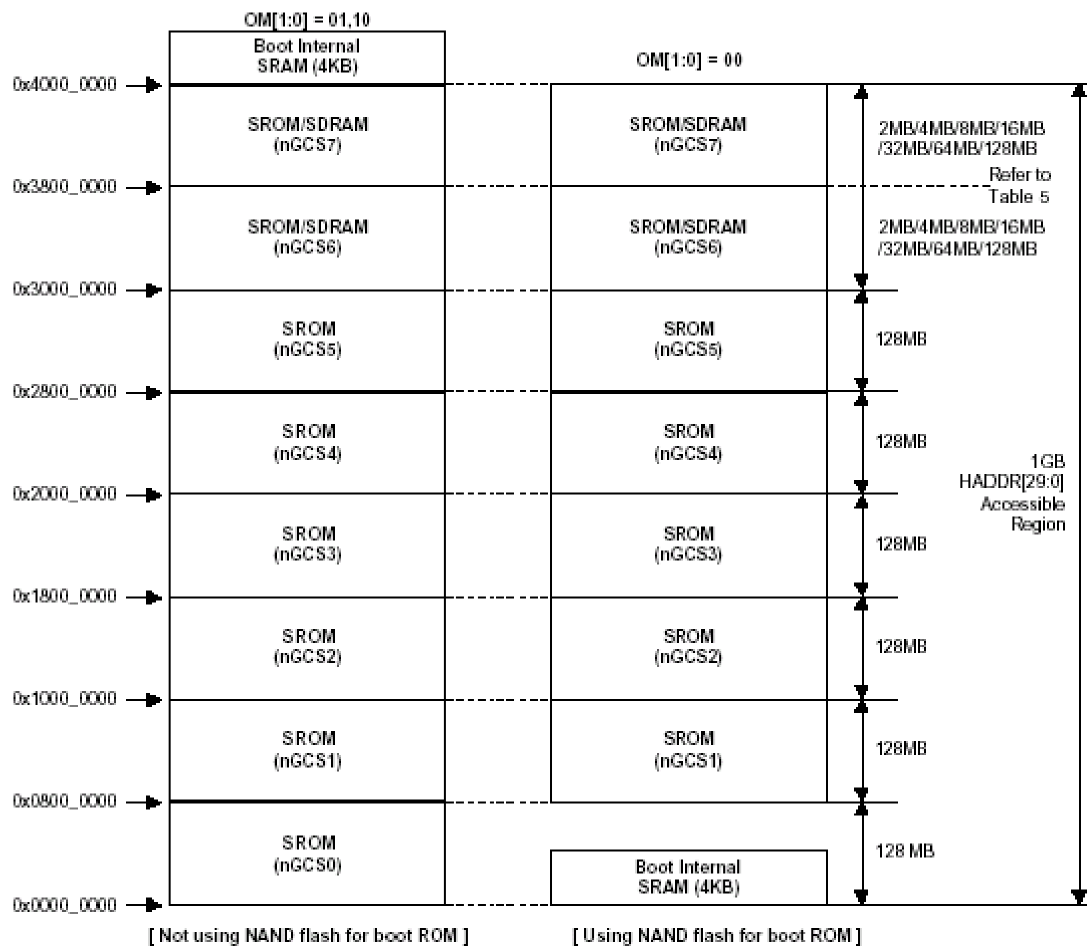
5.2 各个部分的构成

内存部分：

1 片 8 M×16 位数据宽度的 FLASH，共 16M 字节 Flash（intel 28F128J3C，如有不同型号，则是完全兼容的器件），速度 150ns； 两片 16M×16 位数据宽度的 SDRAM（HY57V561620B T，如有不同型号，则是完全兼容的器件）构成，共 64M SDRAM。

5.3 片选

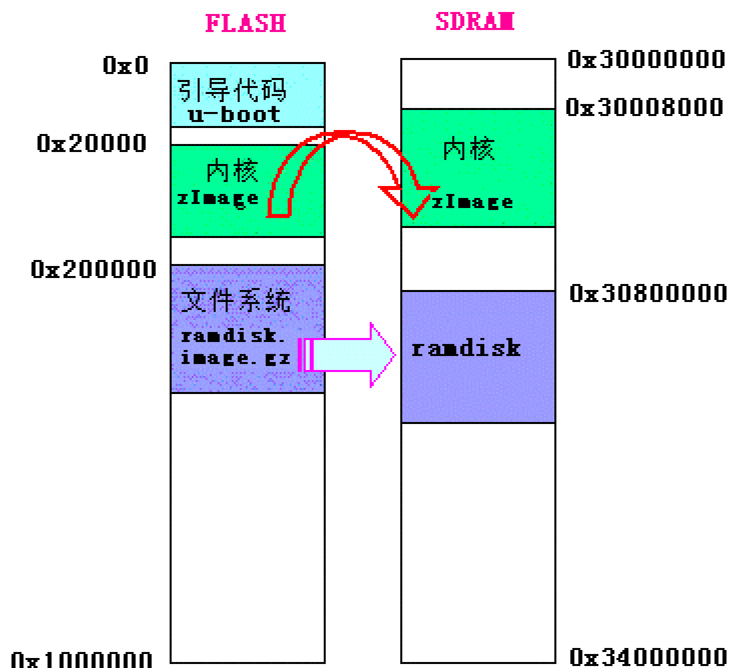
S3C2440 提供 8 路片选，nGCSn[0～7]，每个片选都指定了固定的地址，每个片选固定间隔为 128M 字节。具体参见 CPU 手册 P5-2。



HHARM2440 开发板内存由两片 16M ×16 位数据宽度的 SDRAM 构成，两片拼成 32 位模式，公用 nGCS6。共 64M RAM。起始地址：0x30000000。

nGCS0 接的是一片 8M ×16 位数据宽度的 INTEL E28F128 FLASH，安装在 BANK0，起始地址为 0x0。其中内核 zImage 烧写在地址 0x20000 开始处，根文件系统 RAMDISK 烧在 0x200000 地址处。

下面给出板上的地址空间分布：MEMORY MAP



INTEL E28F128J3A-150 FLASH 的单片 16M 字节，共 128 个扇区，每个扇区都是 128K 字节大小，均匀分布。

5.4 中断

S3C2440 可处理 60 路中断，其中 24 路为外部中断 EINTn，具体可参见 CPU 的 PDF 手册 P14-3。

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC_S/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR, RXD, and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
INT_NFCON	Nand Flash Control Interrupt	ARB4
INT_UART1	UART1 Interrupt (ERR, RXD, and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SDI interrupt	ARB 3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR, RXD, and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2
INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB 2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT_AC97	Watch-Dog timer interrupt(INT_WDT, INT_AC97)	ARB1
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
INT_CAM	Camera Interface (INT_CAM_C, INT_CAM_P)	ARB1
EINT8_23	External interrupt 8 – 23	ARB1
EINT4_7	External interrupt 4 – 7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

INTERRUPT SUB SOURCES

Sub Sources	Descriptions	Source
INT_AC97	AC97 interrupt	INT_WDT_AC97
INT_WDT	Watchdoc interrupt	INT_WDT_AC97
INT_CAM_P	P-port capture interrupt in camera interface	INT_CAM
INT_CAM_C	C-port capture interrupt in camera interface	INT_CAM
INT_ADC_S	ADC interrupt	INT_ADC
INT_TC	Touch screen interrupt (pen up/down)	INT_ADC
INT_ERR2	UART2 error interrupt	INT_UART2
INT_TXD2	UART2 transmit interrupt	INT_UART2
INT_RXD2	UART2 receive interrupt	INT_UART2
INT_ERR1	UART1 error interrupt	INT_UART1
INT_TXD1	UART1 transmit interrupt	INT_UART1
INT_RXD1	UART1 receive interrupt	INT_UART1
INT_ERR0	UART0 error interrupt	INT_UART0
INT_TXD0	UART0 transmit interrupt	INT_UART0
INT_RXD0	UART0 receive interrupt	INT_UART0

板上扩展的外设接口占用片选、中断情况：

接口	占用片选	占用中断
DM9000	nGCS1	EINT0
IDE 硬盘	NGCS3	EINT3
PS/2 键盘	NGCS4	EINT4

在开发板的 minicom 终端可以通过如下命令查看板上的中断信息：

```
# cat proc/interrupts
  0:          1    DM9000 device
 13:          0    DMA timer
 14:       1486    timer
 26:         36    usb-ohci
 52:         31    serial_s3c2440_rx
 53:        100    serial_s3c2440_tx
 54:          0    serial_s3c2440_err
Err:          0
```

5.5 GPIO

S3C2440 提供 130 路复用的 IO 口线，分为如下端口进行管理：

— Port A (GPA): 25路输出口线

— Port B (GPB): 11路输入/输出口线
— Port C (GPC): 16路输入/输出口线
— Port D (GPD): 16路输入/输出口线
— Port E (GPE): 16路输入/输出口线
— Port F (GPF): 8路输入/输出口线
— Port G (GPG): 16路输入/输出口线
— Port H (GPH): 9路输入/输出口线
— Port J(GPJ): 13路输入/输出口线

核心板上将这些复用的信号引脚中未被占用的全部引到底板上来，具体请参见产品光盘中的 PROTEL 格式的电路原理图和 PCB。

5.6 总线

1．S3C2440 是内部 32 位地址，外部 27 位地址，数据总线宽度 32 位。400M 的主频，136M 的总线速度。
2．若外接8 位或16 位数据宽度的外设芯片，与CPU 相接时，HHARM2440的数据总线宽度是可配置的，可分别配为32 位、16 位或8 位模式。设置是在BWSCON 中的DW 位（参见S3C2440X User's Manual的Memory Controller）实现的。在给外设分配片选时，设置好它的BWSCON 中的这两位，在访问它的地址时就可以改变数据宽度。
16 位数据宽度时，是低 16 位数据线有效；
8 位模式时，是最低 8 位数据线有效。

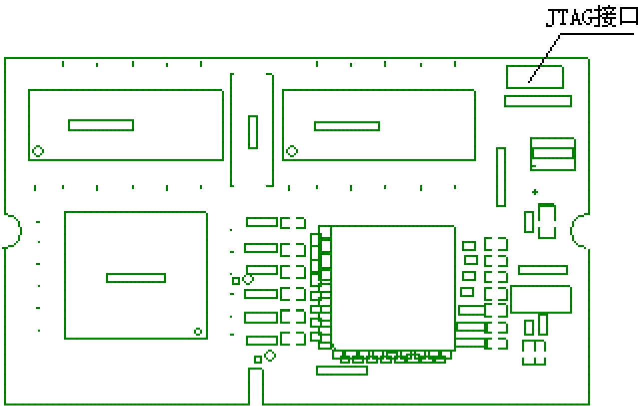
【注意】

启动时这个对 CS0 是无效的，因为 CS0 是接存放启动代码的存贮器片选，一般都是 FLASH，在 CPU 刚加电时，这时的数据宽度就无法用 BWSCON 来设置了，就只有硬件实现了：由复位后硬件配置决定数据的宽度，复位默认为 0x00000000。

两片 SDRAM 为 32 位寻址，但两片的数据总线分别接 HHARM2440 高 16 位和低 16 位数据总线，这样拼成 32 位 SDRAM 使用，所以两片 SDRAM 共享一个 CS。而 一片 FLASH 则固定位 16 位数据读写访问模式，它就只接 HHARM2440 的低 16 位数据总线。

5.7 外设接口图

核心板正面俯视图：



其中核心板上 JTAG 管脚分布如下图示：

2	4	6	8	10
1	3	5	7	9

其对应的管脚定义请参见下面 4.6 节。JTAG 头接 PC 并口一端的有标注的一边为 1 脚。

外设接口底板图：具体参见光盘中的电路图。

5.8 接口管脚说明

1. LED 指示灯

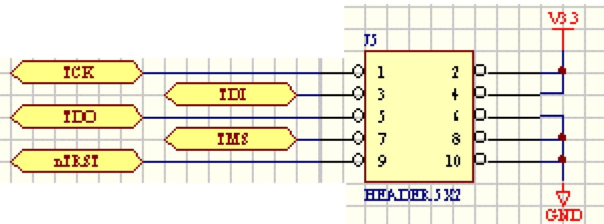
核心板：

状态指示部分：由 LED 提供 3.3V 电源/复位指示。

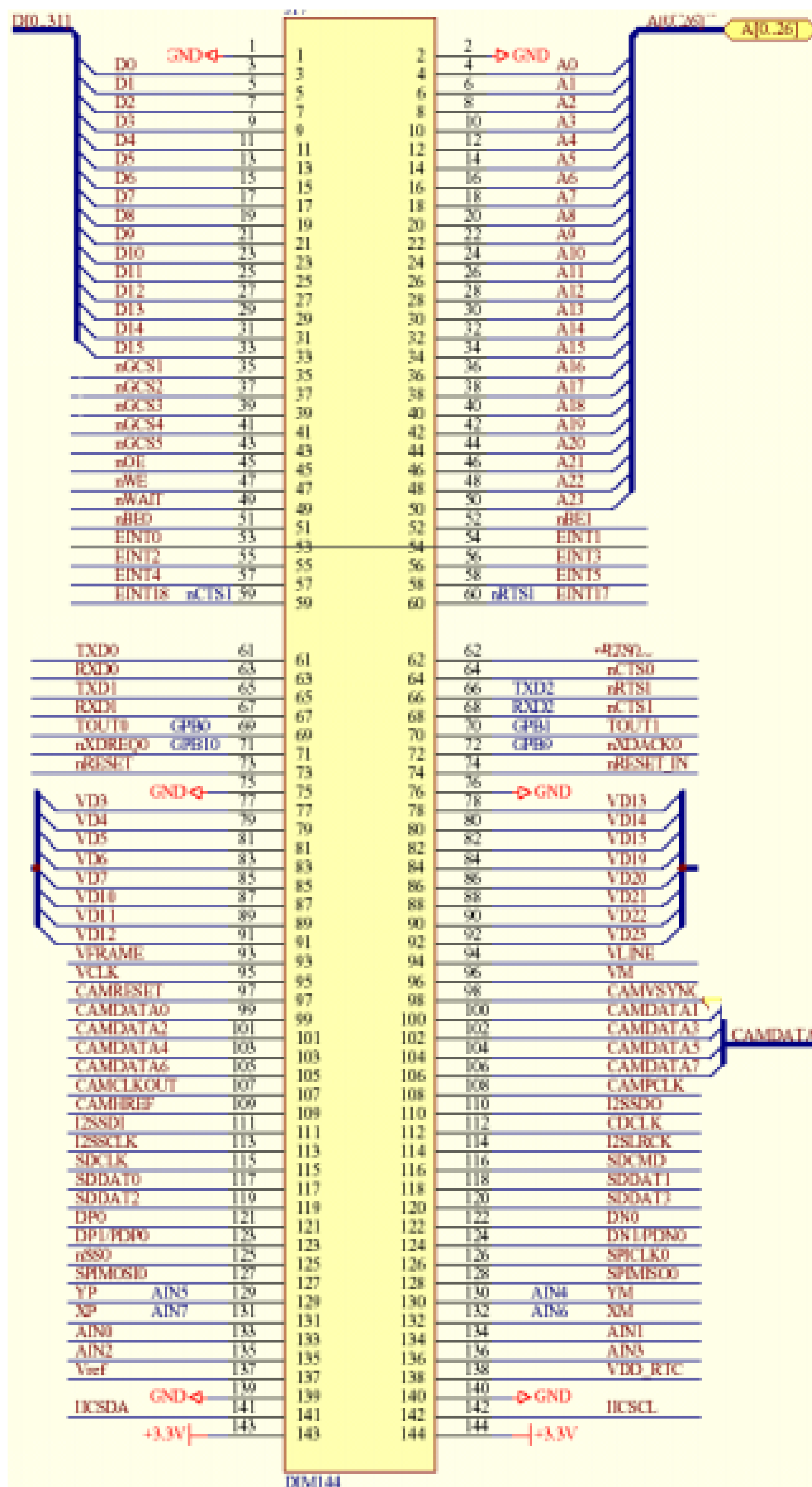
底板（基本板）：

状态指示部分：由 LED 提供电源指示。

2. JTAG 接口



3.DIMM 接口



4． 电气特性

电源	+9V (0 ~ +10%)
工作环境温度(注 1)	0°C ~ 40°C
保藏温度	－20°C ~ 70°C
相对湿度	25%~95%

注 1：可以提供满足-40°C ~ 50°C 工作环境温度的产品；

第六章 底 板 的 硬 件 设 计

6.1 基本板的设计

基本板包括如下主要部分:

与核心板的连接器；

基本端口：包括一个四线串口 COM1；

一个 LCD

一个触摸屏

电源：将外部输入的未稳压电源稳定为核心板和基本板上芯片所需的 3.3V

对于串行口 1 因为在不同的应用中可能会被作为 RS232、RS485/422、红外接口使用，因此对该口必须使用相应的电平转换芯片。在基本板的设计中我们给出的是作为 RS232 端口的方案。

6.2 用户底板原理性设计和硬件方案制定

6.2.1 基本端口的设计

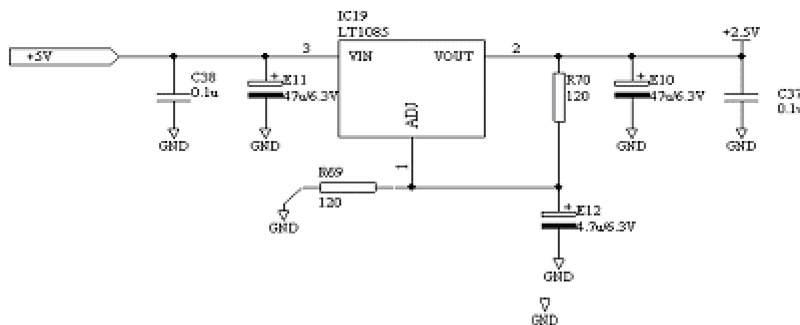
以串口、LCD、触摸屏的设计可以直接参考基本板的设计，具体电路请参考基本板的原理图和 PCB。另外必须注意的是，在串行口电平转换芯片的选择和使用中必须注意电平兼容问题，在基本板中我们使用的是 3.3V 工作电压的 RS232 电平转换芯片，2.5V 工作电压的 Lanswitch 芯片。

6.2.2 电源的设计

核心板工作电源为单一的 3.3V/0.5A 直流，在基本板中，由于电源消耗功率较小，因此我们使用的是 LT10856 线性稳压芯片，使用基本板的上下面铜箔作为散热面，并且使用 9V/0.8A 直流电源供电。

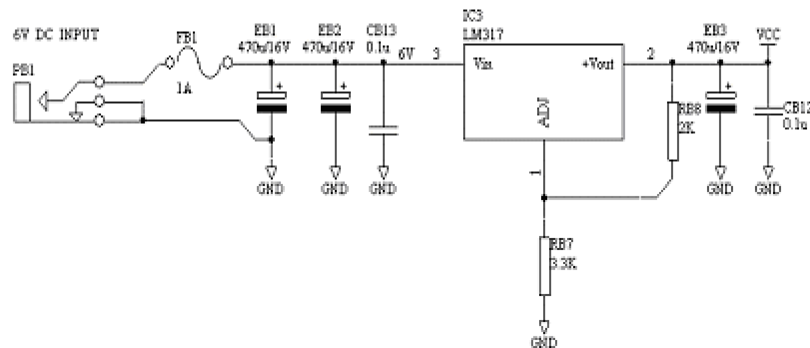
在其他的应用设计中根据不同的电源消耗需求，可以选择线性稳压源方案和开关稳压源方案。对于前一种选择，可以获得低噪声、廉价等益处，但同时也有效率低，发热较大等缺点；对于开关电源方案正好与线性电源的优缺点相反。

使用线性电源时，我们给出以下两种参考：
其一参见下图：

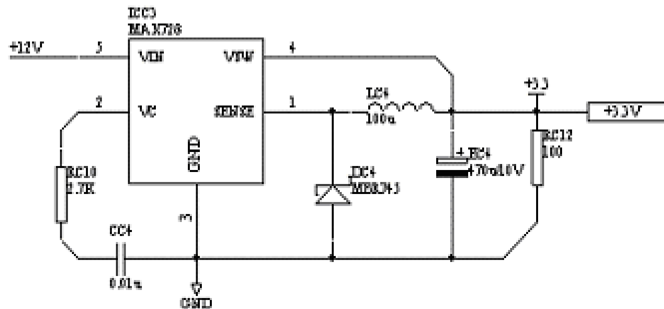


这是一个使用低压差线性稳压器获得 2.5V 稳压输出的例子，输出电压的计算可以使用以下的公式：Vo=1.25V * (R69+R70)/R70；芯片上的耗散功率可以使用以下公式计算：Pc=(Vi-Vo)*Io，由上可以看到，除了输出电流，输入和输出间的压差正比于芯片上的耗散功率。稳压芯片的金属背板表面工作温度必须小于 80 摄氏度，设其金属背板到散热器的热阻为 Rt1，散热器到空气的热阻为 Rt2，则稳压芯片金属背板与环境的温升为：.T=Pc/(Rt1+Rt2)；由此可以看到减少温升必须减小 Pc 或者总热阻，后者可以通过减小 Rt1（使用导热硅脂于散热器于管壳之间），减少 Rt2（增大散热器面积）；

其二参见下图：



这是一个使用最廉价的普通线性稳压器获得 3.3V 稳压输出的例子，输出电压的计算可以使用以下的公式：Vo=1.25V ＊ (RB8+RB7)/RB8；其他同上，但要注意的是 LM317 必须有大于 2.5V 的输入输出压差才能工作在稳压状态。因此最下输入电压要大于上者。
使用开关电源时，我们给出以下参考：
见下图：



要注意的是电感 LC4 的选择，其最小平均直流工作电流必须大于输出电流（既在规定的输出电流下其不能出现磁芯饱和现象）。另外必须注意 PCB 的排版，这将决定电源的噪声幅度，在以上电路中 EC4 为钽电解电容，在一般的应用中必须另外附加并联一个 0.1u 的瓷介电容,以滤除高频分量。电源类型的选择，最终是效率、成本、体积等因素的综合考虑。

6.2.3 其它电路部分的设计

在这里特别指出应当充分重视对核心板各个引出线添加缓冲电路，以充分保证核心板和处理器的稳定性；对于核心板连接器的地线和电源线引脚应当全部连接到相应的地和电源上。

6.2.4 PCB 设计和排版时的注意事项

PCB 布局时的考虑

首先由于 144PIN 双列脚比较密，无法在焊盘间穿线，在核心板设计中将上述插座的外围引脚大部分作为电源或地线。其次由于核心板的读写访问速率较高，因此必须考虑到布线传输延迟对电路的影响。在底板 PCB 走线设计中时应当将涉及到数据总线、地址总线、内存访问控制信号和中断信号等的电路部分排布在 CN1,CN2 连接器之间,以方便布线和减少信号延迟。简单应用时可以使用 2 层板，并且大面积铺地；复杂的应用使用多层布线，并且至少有一层为地层（最好将相邻的层作为电源层。当板上有数个电源品种时，一般来讲如果可以将使用同种电源的电路部分布局在临近区域内，然后将电源层分割为数个区块。）

另外还应当注意以下事项：

核心板接口插座的整体布局；

板上提供必要的测试焊盘；

6.2.5 PCB 在电路稳定性和抗干扰方面的考虑

应重点考虑以下几点：

1. 合理的地层和电源层分割

建议使用较为完整的一个 PLANE 作为地层，如果存在多个“地”(例如：数字地，模拟地，保护地等）可以在同一个地层上适当分割获得。对于电源层类似上述，并且应当注意低噪声电路部分电源和高噪声电路部分电源的隔离或滤波。

2. 合理的去耦电容排布

在电源进线端，在各个数字芯片电源引脚附近等分布适当的去耦电容。对核心板的隔离，以及在各个端口使用适当的输入输出缓冲；

附　　录

附录 A LINUX 常见术语

Linux

“Linux 是一种 UNIX 操作系统的克隆，它（的内核）由 Linus Torvalds 以及网络上组织松散的黑客队伍一起从零开始编写而成。 Linux 的目标是保持和 POSIX 的兼容。

“Linux 具备现代一切功能完整的 UNIX 系统所具备的全部特征，其中包括真正的多任务、虚拟内存、共享库、需求装载、共享的写时复制程序执行、优秀的内存管理以及 TCP/IP 网络支持等。

“Linux 的发行遵守 GNU 的通用公共许可证。

“Linux 起初为基于 386/486 的 PC 机开发，但现在，Linux 也可以运行在 DEC Alpha、SUN Sparc、M68000，以及 MIPS 和 PowerPC 等计算机上。”

POSIX

POSIX 表示可移植操作系统接口（Portable Operating System Interface ，缩写为 POSIX 是为了读音更像 UNIX）。电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。许多其它的操作系统，例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990（1995 年修订）或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。

POSIX 现在已经发展成为一个非常庞大的标准族，某些部分正处在开发过程中。POSIX 与 IEEE 1003 和 2003 家族的标准是可互换的。

GNU

GNU 是 GNU Is Not UNIX 的递归缩写，是自由软件基金会的一个项目，该项目的目标是开发一个自由的 UNIX 版本，这一 UNIX 版本称为 HURD。尽管 HURD 尚未完成，但 GNU 项目已经开发了许多高质量的编程工具，包括 emacs 编辑器、著名的 GNU C 和 C++ 编译器（gcc 和 g++），这些编译器可以在任何计算机系统上运行。所有的 GNU 软件和派生工作均适用 GNU 通用公共许可证，即 GPL。GPL 允许软件作者拥有软件版权，但授予其他任何人以合法复制、发行和修改软件的

权利。

Linux 的开发使用了许多 GNU 工具。Linux 系统上用于实现 POSIX.2 标准的工具几乎都是 GNU 项目开发的，Linux 内核、GNU 工具以及其他一些自由软件组成了人们常说的 Linux：

符合 POSIX 标准的操作系统 Shell 和外围工具。

C 语言编译器和其他开发工具及函数库。

X Window 窗口系统。

各种应用软件，包括字处理软件、图象处理软件等。

其他各种 Internet 软件，包括 FTP 服务器、WWW 服务器等。

关系数据库管理系统等。

GPL

GPL (General Public License)文本保存在 Linux 系统的不同目录下的命名为 COPYING 的文件里。例如，键入 cd /usr/doc/ghostscript* 然后再键入 more COPYING 可查看 GPL 的内容。

GPL 和软件是否免费无关，它主要目标是保证软件对所有的用户来说是自由的。GPL 通过如下途径实现这一目标：

它要求软件以源代码的形式发布，并规定任何用户能够以源代码的形式将软件复制或发布给别的用户。

它提醒每个用户，对于该软件不提供任何形式的担保。如果用户的软件使用了受 GPL 保护的任何软件的一部分，那么该软件就继承了 GPL 软件，并因此而成为 GPL 软件，也就是说必须随应用程序一起发布源代码。

GPL 并不排斥对自由软件进行商业性质的包装和发行，也不限制在自由软件的基础上打包发行其他非自由软件。遵照 GPL 的软件并不是可以任意传播的，这些软件通常都有正式的版权，GPL 在发布软件或者复制软件时声明限制条件。但是，从用户的角度考虑，这些根本不能算是限制条件，相反用户只会从中受益，因为用户可以确保获得源代码。

尽管 Linux 内核也属于 GPL 范畴，但 GPL 并不适用于通过系统调用而使用内核服务的应用程序，通常把这种应用程序看作是内核的正常使用。

Linux 的主要发行版

Red Hat Linux 采用 RPM 的软件保管理方式，软件的安装、卸载和升级非常方便，并提供了大量的图形化管理工具，是初学者的最佳选择。

Mandrake

Slackware

Debian GNU/Linux

是由 GNU 发行的 Linux 版本，最符合 GNU 精神。提供了最大的灵活性，适合 Linux 的高级用户。

附录 B 常用 LINUX 命令

以下均以 REDHAT LINUX 为例说明。

基本命令：

ls：显示当前目录下的所有文件和目录。

ls –a ：可以看到隐藏的文件，如以.开头的文件。

pwd：显示当前目录路径。

ps：列举当前 TTY 下所有进程

ps –A：列举所有

cd 目录名：进入目录

mkdir 目录名：创建目录

rmdir 目录名：删除空目录

rm –rf 目录名：强行删除整个目录内容（无法恢复），其中 f 表示强制不进行提示，r 表示目录递归。

LINUX 下的文件和目录是区分大小写的。

TAB 文件目录匹配搜索的使用

例如华恒软件安装的目录为：/HHARM2440，假设/目录下没有其它以 HHA 字符开头的其它目录和文件，则要进入这个目录，只需敲入：

cd /HHA

然后按下 TAB 键，则 SHELL 会自动匹配找到/HHARM2440 目录，这样就不必完全键入剩余的 RM2440 字符，这个功能在访问名字很长的文件和目录时非常有效，可以大大提供键盘输入的速度，极为方便。

ncftp 工具的使用：

ncftp 是 LINUX 下非常好的 FTP 工具软件，它除了支持 FTP 命令操作外，还支持 LINUX SHELL 下的命令用法，例如，它也支持 TAB 键用法，支持目录上传和下载(用-r 或-R 参数)。ncftp 的用法,例如要 FTP 一台 IP 为 192.168.2.126 的 LINUX PC 机 A，命令如下：

ncftp –u hhcn 192.168.2.126

其中 hhcn 为 A 机器上的合法的用户，连接上之后会提示输入 hhcn 用户的密码，密码验证通过后，就进入 ncftp 命令提示符。

编程时获取帮助 man（类似于 VC 编程中的 MSDN）

man，即 manunal，是 UNIX 系统手册的电子版本。根据习惯，UNIX 系统手册通常分为不同的部分（或小节，即 section），每个小节阐述不同的系统内容。目前的小节划分如下：

命令：普通用户命令

系统调用：内核接口
函数库调用：普通函数库中的函数
特殊文件：/dev 目录中的特殊文件
文件格式和约定：/etc/passwd 等文件的格式
游戏。
杂项和约定：标准文件系统布局、手册页结构等杂项内容
系统管理命令。
内核例程：非标准的手册小节。

手册页一般保存在 /usr/man 目录下，其中每个子目录（如 man1, man2, ..., manl, mann）包含不同的手册小节。使用 man 命令查看手册页。
常用 man 命令行：
man strtoul

取消 root 密码：

vim /etc/shadow

可以看到第一行内容大致如下：

root:1dVVd5YVP$OgZG58TL/NRExTfcr6URH.:11829:0:99999:7:-1:-1:134539236

要取消 root 密码，只需将第一行 root 后第一对:之间的字符全部删除即可，删除后如下：

root::11829:0:99999:7:-1:-1:134539236

然后用:w!强行存盘（因为 shadow 文件是只读的）后用:q 退出 vi 则实现取消了 root 密码。

修改 PC 机 IP 地址：

ifconfig eth0 192.168.2.126

压缩/解压缩：

LINUX 的软件一般是以.gz 或.tar 或者.tar.gz 结尾的。前者是由 gzip 压缩的，后者是先用 tar 归档，在用 gzip 压缩而成的。

1、以.gz 结尾的为压缩文件，用命令：gzip -d filename 来解压，得到的文件在当前目录中，但已没有了.gz。

2、以.tar 结尾的为归档文件，用命令：tar -xvf filename 来展开，生成的文件与源文件在同一目录中，只是少了.tar。

3、以.tar.gz 结尾的文件最常见，可直接用命令：gzip -cd filename | tar xfv 来解开。

tar 的用法：
解压：x 参数表示解压

tar xzf hharm2440.tgz

把一个目录 HHARM2440 压缩成一个文件：hhHHARM2440.tgz

tar czf hharm2440.tgz HHARM2440

c 参数表示压缩。

查找文件，如：main.c：

find –name main.c

或者：

locate shadow

【注意】:locate 为模糊匹配,它会递归的在当前目录下的所有子目录下搜索，并列出所有名字包含 shadow 字串的文件。

在一个目录下（含子目录）的所有文件中查找含有某个字符串（如“ Modified by hhcn ”）的所有文件：

grep ‘Modified by hhcn’ * -r

ctags 的用法：

使用“ ctags –R * ”命令先在当前目录下建立一个 tags 文件，那么只要在 源代码目录下由 vim 去开启文件的话，会自动加载 tags ,这个文件，无需另行加载，否则要由 :set tags=your.tags 来指定 tags 文件。然后就是照一般使用 Vim ，光标移到标识符或函数名上，按 Ctrl+]，要回到原处就按 Ctrl+T。

请注意，Vim 激活时，工作目录（vim 激活时的所在目录）名为 tags 的文件会自动加载，$VIMRUNTIME/doc 及 $HOME/.vim/doc

目录下的 tags 文件也会自动加载。而且，凡是加载的 tags 文件里头所有标志文字都可以使用补全键来补全，别忘了这个好用的功能。

vi(m)用法

vi 是 Linux/Unix 世界里极为普遍的全屏幕文本编辑器，几乎可以说任何一台 Linux/Unix 机器都会提供这个软件。

vi 有三种状态，即编辑方式、插入方式和命令方式。

在命令方式下，所有命令都要以:开始，所键入的字符系统均作命令来处理，如:q 代表退出，:w 表示存盘。

当你进入 vi 时，会首先进入命令方式（同时也是编辑方式）。按下 I 就进入插入方式，用户输入的可视字符都添加到文件中，显示在屏幕上。按下 ESC 就可以回到命令状态（同时也是编辑方式）。

编辑方式和命令方式类似，都是要输入命令，但它的命令不要以:开始，它直接接受键盘输入的单字符或组合字符命令，例如直接按下 u 就表示取消上一次对文件的修改，相当于 WINDOWS 下的 Undo 操作。编译方式下有一些命令是要以/开始的，例如查找字符串就是 :/string 则在文件中匹配查找 string 字符串。在编辑模式下按下:就进入命令方式。

基本命令解释：

1．光标命令

k、j、h、l——上、下、左、右光标移动命令。虽然您可以在 Linux 中使用键盘右边的 4 个光标键，但是记住这 4 个命令还是非常有用的。这 4 个键正是右手在键盘上放置的基本位置。

nG——跳转命令。n 为行数，该命令立即使光标跳到指定行。

Ctrl+G——光标所在位置的行数和列数报告。

w、b——使光标向前或向后跳过一个单词。

2．编辑命令

i、a、r——在光标的前、后以及所在处插入字符命令(i=insert、a=append、r=replace)。

cw、dw——改变(置换)/删除光标所在处的单词的命令 (c=change、d=delete)。

x、d$、dd——删除一个字符、删除光标所在处到行尾的所有字符以及删除整行的命令。

3．查找命令

---- /string、?string——从光标所在处向后或向前查找相应的字符串的命令。

4．拷贝复制命令

---- yy、p——拷贝一行到剪贴板或取出剪贴板中内容的命令。

常用操作：

无论是开启新档或修改旧文件，都可以使用 vi，所需指令为：

$ vi filemane

如果文件是新的，就会在荧幕底部看到一个信息，告诉用户正在创建新文件。如果文件早已存在，vi 则会显示文件的首廿四行， 用户可再用光标（cursor）上下移动。

～

～

上面是一个经 vi 开启的模拟文件，一行开始处的波折号（～）表示文件的结尾。

—指令 i 在光标处插入正文

—指令 I 在一行开始处插入正文

—指令 a 在光标後追加正文

—指令 A 在行尾追加正文

—指令 o 在光标下面新开一行

—指令 O 在光标上面新开一行

在插入方式下，不能打入指令，必需先按〈Esc〉键，返回命令方式。假若户不知身处何态，也可以按〈Esc〉键，不管处於何态，都会返回命令方式。

在修改文件时，如何存档及退出指定文件都非常重要。在 vi 内，行使存档或退出的指令时，要先按冒号（:），改变为命令方式，用户就可以看见在荧幕左下方，出现冒号（:），显示 vi 已经改为指 令态，可以进行存档或退出等工作。

:q!放弃任何改动而退出 vi，也就是强行退出

:w 存档

:w! 对于只读文件强行存档

:wq 存档并退出 vi

:x 与 wq 的工作一样

:zz 与 wq 的工作一样删除正文

删除或修改正文都是利用编辑方式，故此，下面所提及的指令只需在编辑方式下，直接键入指令即行。

—x 删除光标处字符（Character）

—nx 删除光标处後 n 个字符

—nX 删除光标处前 n 个字符

—ndw 删除光标处下 n 个单词（word）

—dd 删除整行

—d$或 D 删除由光标至该行最末

—u 恢复前一次所做的删除

当使用 vi 修改正文，加减字符时，就会采用另一组在编辑方式下操作的指令。

— r char 由 char 代替光标处的字符

—Rtext〈Esc〉由 text 代替光标处的字符

—cwtext〈Esc〉由 text 取代光标处的单词

—Ctext〈Esc〉由 text 取代光标处至该行结尾处

—cc 使整行空白，但保留光标位置，让你开始打入

—如删除指令一样，在指令前打入的数，表示执行该指令多少次。

要检索文件，必需在编辑方式下进行。

—／str〈Return〉向前搜寻 str 直至文件结尾处

—?str〈Return〉往後搜寻 str 直至文件开首处

—n 同一方向上重复检索
—N 相反方向上重复检索
—vi 缠绕整个文件，不断检索，直至找到与模式相匹配的下一个出现。

全程替换命令：
:%s/string1/string2/g 在整个文件中替换“string1”成“string2”。
如果要替换文件中的路径：
使用命令“:%s#/usr/bin#/bin#g”可以把文件中所有路径/usr/bin 换成/bin。也可以使用命令“:%s/\/usr\/bin/\/bin/g”实现，其中“\”是转义字符，表明其后的“/”字符是具有实际意义的字符，不是分隔符。

同时编辑 2 个文件，拷贝一个文件中的文本并粘贴到另一个文件中：
命令如下：
---- vi file1 file2
---- yy 在文件 1 的光标处拷贝所在行
---- :n 切换到文件 2 (n=next) 或者按 ctrl+ww，就在两个文件间切换。
---- p 在文件 2 的光标所在处粘贴所拷贝的行
---- :n 切换回文件 1

将文件中的某一部分修改保存到临时文件，例如仅仅把第 20～59 行之间的内容存盘成文件/tmp/1，我们可以键入如下命令。
---- vi file
---- :20,59w /tmp/1

如果要在 vi 执行期间，转到 shell 执行，使用惊叹号（!）执行系统指令，例如在 vi 期间，列出当前目录内容，可以键入 ：
:!ls
另一方面，用户可以在主目录中创建.exrc 环境文件，用 set 打入选项，每次调用 vi 时，就会读入.exrc 中的指令与设置。下面是.exrc 环境文件的实例：
```
set wrapmarging=8
set showmode
set autoindent
```
minicom 用法
minicom 是安装 REDHAT 时安装的软件，它使用配置文件/etc/minirc.dfl，华恒光盘安装时会提供这个文件。
【注意】
minicom 占用串口，能且仅能启动一个 minicom，启动第二个时就会报错：

Device /dev/modem is locked。其中/dev/modem 就是/dev/ttyS0，即 PC 机串口 1，它是在光盘安装时执行./arminst 时创建的链接。查看 arminst 文件，可以看到如下一行：

```
ln -sf /dev/ttyS0 /dev/modem
```

minicom 所有的操作都以 ctrl+A 开始，例如：退出为 ctrl+A，松手后再按下 Q，则弹出如下一个小框：选 Yes 即可退出 minicom。

```
Leave without reset?
   Yes        No
```

minicom 中最重要的操作就是对其进行配置的修改。这个操作要先 ctrl+A，松手后按下 O，则弹出如下框：

```
[configuration]
 Filenames and paths
 File transfer protocols
 Serial port setup
 Modem and dialing
 Screen and keyboard
 Save setup as dfl
 Save setup as..
 Exit
```

选择第三项“Serial port setup”，则弹出下面框：

```
A -    Serial Device      : /dev/modem
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       :115200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

   Change which setting?
```

键入 E 则弹出如下框，可改变波特率：

```
-----------------[Comm Parameters]-----------------

 Current:115200 8N1

   Speed              Parity            Data

 A: 300               L: None           S: 5
 B: 1200              M: Even           T: 6
 C: 2400              N: Odd            U: 7
 D: 4800              O: Mark           V: 8
 E: 9600              P: Space
 F: 19200
 G: 38400
 H: 57600
 I: 115200            Q: 8-N-1
 J: 230400            R: 7-E-1


 Choice, or <Enter> to exit?
```

若要使用 PC 机的串口 2 来接开发板的串口 1 做监控，则要在串口配置框中选择 A，即“ Serial Device ”，则原来的配置框第一行进入编辑模式，将原来的/dev/modem 改为如下的：/dev/ttyS1，即串口 2。

```
A -     Serial Device      : /dev/ttyS1
B - Lockfile Location      : /var/lock
C -    Callin Program      :
D -   Callout Program      :
E -     Bps/Par/Bits       :115200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

    Change which setting?
```

退出配置框只需连续按 ESC 键即可返回。

软、硬盘及光驱的使用

在 Linux 中对其他硬盘逻辑分区、软盘，光盘的使用与我们通常在 DOS 与 Windows 中的使用方法是不一样的，不能直接访问，因为在 Linux 中它们都被视为文件，因此在访问使用前必须使用装载命令 mount 将它们装载到系统的/mnt 目录中来，使用结束，必须进行卸载。命令格式如下：

mount –t 文件系统类型 设备名 装载目录

文件类型常用的有：

msdos　　dos 分区文件
ext2　　Linux 的文件系统
swap　　Linux swap 分区或 swap 文件
iso9660　　安装 CD-ROM 的文件系统
vfat　　支持长文件名的 dos 分区
hpfs　　OS/2 分区文件系统

设备名是指要装载的设备的名称，如软盘、硬盘、光盘等，软盘

一般为/dev/fd0 fd1,硬盘一般为/dev/hda hdb ,硬盘逻辑分区一般为期 hda1 hda2...等等，光盘一般为/dev/hdc。在装载前一般要在/dev/mnt 目录下建立一个空的目录，如软盘为 floppy，硬盘分区为其盘符如 c、d 等等，光盘为 cd-rom，使用命令：

mount –t msdos /dev/fd0 /mnt/floppy

装载一个 mddos 格式的软盘

mount –t ext2 /dev/fd0 /mnt/floppy

装载一个 Linux 格式的软盘

mount –t vfat /dev/hda1 /mnt/c

装载 Windows98 格式的硬盘分区

mount –t iso9660 /dev/hdc /mnt/cd-rom

装载一个光盘

装载完成之后便可对该目录进行操作，在使用新的软盘及光盘前必须退出该目录，使用卸载命令进行卸载，方可使用新的软盘及光盘，否则系统不会承认该软盘的，光盘在卸载前是不能用光驱面板前的弹出键退出的。

LILO 与 GRUB

LINUX 安装时一般都安装 bootloader，可以支持多操作系统并存。常见的 bootloader 有 LILO 和 GRUB，REDHAT6.x 使用 LILO，REDHAT7.2 同时支持 LILO 和 GRUB，但默认使用的是 GRUB。

Diff 创建软件补丁，用 patch 打补丁

diff 是生成源码补丁的必备工具。其命令格式为：

diff [命令行选项] 原始文件 新文件

常用命令行选项如下：

-r 递归处理目录　　　　-u 输出统一格式(unified format)
-N patch 里包含新文件　　-a patch 里可以包含二进制文件

它的输出在 stdout 上，所以你可能需要把它重定向到一个文件。

输出格式保存了上下文（缺省是上下各三行，最少需要两行），这样，patch 的时候可以允许行号不精确匹配的情况出现。另外，在 patch 文件的开头明确地

用---和+++标示出原始文件和当前文件，也方便阅读。

通常，我们需要对整个软件包做修改，并生成一个 patch 文件，下面是典型的操作过程。

```
tar xzvf software.tar.gz # 展开原始软件包，其目录为 software
cp _a software software-orig # 做个修改前的备份
cd software
[修改，测试……]
cd ..
diff -ruNa software-orig software > software-my.patch
```

现在我们就可以保存 software-my.patch 做为这次修改的结果，至于原始软件包，可以不必保存。等到下次需要再修改的时候，可以用 patch 命令把这个补丁打进原始包，再继续工作。比如是在 Linux kernel 上做的工作，就不必每次保存几十兆修改后的源码了。这是好处之一，好处之二是维护方便，由于 unified patch 格式有一定的模糊匹配能力，能减少原软件包升级带来的维护工作量。

patch

patch 程序根据补丁（patch）文件修改一个文件。补丁文件通常是使用 diff 程序建立的一个列表，这个列表包含如何修改原始文件的一些指令。由于节省时间和空间，Patch 经常用于源代码的修补。可以想象一个有 1MB 的程序包，这个程序包的下一个版本仅仅改变了前面一个版本的两个文件的内容，这个新版本可以完全以一个 1MB 的新版本进行发布或者以一个仅仅有 1KB 的补丁文件进行发布。这个补丁文件可以对第一个版本的进行更新，更新后的版本就和第二个版本完全一致了。因此，如果已经下载了第一个版本，那么为了下一个版本而进行的大数据量下载工作就可以有效地避免。

常用命令行选项：

patch [命令行选项] [待 patch 的文件[patch]]

-pn patch level(n 是数字) -b[后缀] 生成备份，缺省是.orig

为了说明什么是 patch level，这里看一个 patch 文件的头标记。

```
diff -ruNa xc.orig/config/cf/Imake.cf xc.bsd/config/cf/Imake.cf
--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999
+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000
```

这个 patch 如果直接应用，它会去找 xc.orig/config/cf 目录下的 Imake.cf 文件，假如你的源码树的根目录是缺省的 xc 而不是 xc.orig，除了 mv xc xc.orig 之外，有无简单的方法应用此 patch 呢？patch level 就是为此而设：patch 会把目标路径名砍去开头 patch level 个节(由/分开的部分)。在本例中，可以用下述命令：

```
cd xc; patch -p1 < /pathname/xxx.patch
```

完成操作。注意，由于没有指定 patch 文件，patch 程序默认从 stdin 读入，所以用了输入重定向。

又例如：

```
diff -r dir1 dir2 >patch20020523.patch
```

递归的比较目录 dir1 与 dir2 内，所有各文件之不同处，并将不同处记录到 patch20020523.patch 文件中。

```
patch -p1 < [patchfile]
```

-p1 选项代表 patchfile 中文件名左边目录的层数，顶层目录在不同的机器上有所不同。要使用这个选项，就要把你的 patch 放在要被打补丁的目录下，然后在这个目录中运行 path -p1 < [patchfile]。

LINUX 下的硬盘分区

对习惯于使用 Dos 或 Windows 的用户来说，有几个分区就有几个驱动器，并且每个分区都会获得一个字母标识符，然后就可以选用这个字母来指定在这个分区上的文件和目录，它们的文件结构都是独立的，非常好理解。但对 Red Hat Linux 用户来说无论有几个分区，分给哪一目录使用，它归根结底就只有一个根目录，一个独立且唯一的文件结构。Red Hat Linux 中每个分区都是用来组成整个文件系统的一部分，因为它采用了一种叫“载入”的处理方法，它的整个文件系统中包含了一整套的文件和目录，且将一个分区和一个目录联系起来。这时要载入的一个分区将使它的存储空间在一个目录下获得。

对于 IDE 硬盘，驱动器标识符为“hdx~”，其中“hd”表明分区所在设备的类型，这里是指 IDE 硬盘了。“x”为盘号（a 为基本盘，b 为基本从属盘，c 为辅助主盘，d 为辅助从属盘），“~”代表分区，前四个分区用数字 1 到 4 表示，它们是主分区或扩展分区，从 5 开始就是逻辑分区。例，hda3 表示为第一个 IDE 硬盘上的第三个主分区或扩展分区，hdb2 表示为第二个 IDE 硬盘上的第二个主分区或扩展分区。对于 SCSI 硬盘则标识为“sdx~”，SCSI 硬盘是用“sd”来表示分区所在设备的类型的，其余则和 IDE 硬盘的表示方法一样。

从上面可以看到，Red Hat Linux 的分区是不同于其它操作系统分区的，它的分区格式只有 Ext2（3）和 Swap 两种，Ext2（3）用于存放系统文件，Swap 则作为 Red Hat Linux 的交换分区。Red Hat Linux 至少需要两个专门的分区（Linux Native 和 Linux Swap）况且不能将 Red Hat Linux 安装在 Dos/Windows 分区。一般来说将 Red Hat Linux 安装一个或多个类型为“Linux Native”的硬盘分区，但是在 Red Hat Linux 的每一个分区都必须要指定一个“Mount Point”（载入点），告诉 Red Hat Linux 在启动时，这个目录要给哪个目录使用。对“Swap”分区来说，一般定义一个且它不必要定义载入点。

SWAP 分区是 LINUX 暂时存储数据的交换分区，它主要是把主内存上暂时不用的数据存起来，在需要的时候再调进内存内，且作为 SWAP 使用的分区不用指定

“Mout Point”（载入点），既然它作为交换分区，我们理所当然应给它指定大小，它至少要等于系统上实际内存的量，一般来说它的大小是内存的两倍，如果你是 16MB 的内存，那么 SWAP 分区的大小是 32MB 左右，以此类推。但必须还要注意一点，SWAP 分区不要大于 128MB，如果你是 64MB 的内存，那么 SWAP 分区最大也只能被定为 127MB，再大就是浪费空间了，因为系统不需要太大的交换分区。以此类推，如果你是 128MB 或更大的内存，SWAP 分区也只能最大被定为 127MB。可以创建和使用一个以上的交换分区，最多 16 个。

Linux Native 是存放系统文件的地方，它只能用 EXT2（3）的分区类型。对 Windows 用户来说，操作系统必须装在同一分区里，它是商业软件吗!所以你没有选择的余地！对 Red Hat Linux 来说，你有了较大的选择余地，你可以把系统文件分几个区来装（必须要说明载入点），也可以就装在同一个分区中（载入点是“/”）。

/boot 分区，它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是有必要的，因为目前大多数的 PC 机要受到 BIOS 的限制,况且如果有了一个单独的/boot 启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在 50MB—100MB 之间。但是如果想用 LILO 启动 Red Hat Linux 系统的话,含有/boot 的分区必须完全在柱面 1023 以下。又由于 8GB 后的数据 LILO 不能读取，所以 Red Hat Linux 要安装在 8GB 的区域以内。

/usr 分区，是 Red Hat Linux 系统存放软件的地方，如有可能应将最大空间分给它。

/home 分区，是用户的 home 目录所在地，这个分区的大小取决于有多少用户。如果是多用户共同使用一台电脑的话，这个分区是完全有必要的，况且根用户也可以很好地控制普通用户使用计算机，如对用户或者用户组实行硬盘限量使用，限制普通用户访问哪些文件等。其实单用户也有建立这个分区的必要，因为没这个分区的话,那么你只能以 root 用户的身份登陆系统,这样做是危险的,因为 root 用户对系统有绝对的使用权，一旦对系统进行了误操作，就会导致系统崩溃。

/var/log 分区，是系统日志记录分区，如果设立了这一单独的分区，这样即使系统的日志文件出现了问题，它们也不会影响到操作系统的主分区。

/tmp 分区，用来存放临时文件。这对于多用户系统或者网络服务器来说是有必要的。这样即使程序运行时生成大量的临时文件，或者用户对系统进行了错误的操作，文件系统的其它部分仍然是安全的。因为文件系统的这一部分仍然还承受着读写操作，所以它通常会比其它的部分更快地发生问题。

/bin 分区，存放标准系统实用程序。

/dev 分区，存放设备文件。

/opt 分区，存放可选的安装的软件。

/sbin 分区，存放标准系统管理可执行文件，如 insmod，ifconfig 等。

上面介绍了几个常用的分区，一般来说需要一个 SWAP 分区，一个/boot 分区，

一个/usr 分区，一个/home 分区，一个/var/log 分区。当然这没有什么规定，完全是依照个人来定的。但记住至少要有两个分区，一个 SWAP 分区，一个/分区。用户可以使用两种分区工具：

1. Disk Druid：它是 Red Hat Linux 提供的硬盘管理工具，它最初是随 Red HatLinux5 一起发售的，它可以根据用户的要求创建和删除硬盘分区，另外还可以为每个分区管理载入点，这是一个不错的分区软件，建议读者使用。本文也将以此软件详细地介绍 Red Hat Linux 分区。

2. Fdisk：它是传统的 Linux 硬盘分区工具，比 Disk Druid 更强大，使用更加灵活。但是 Fdisk 要求用户对硬盘分区有一定经验，并能够适应且读懂简单的文本界面。如果你是第一次对一个硬盘驱动器进行分区操作的话，最好还是避免 Fdisk 这样的程序，它虽然强大但用起来的感觉不是太好的。

附录 C gcc 与 gdb

gcc 是 GNU 的 C 和 C++ 编译器，它是 Linux 中最重要的软件开发工具。实际上，gcc 能够编译三种语言：C、C++ 和 Object C（C 语言的一种面向对象扩展）。利用 gcc 命令可同时编译并连接 C 和 C++ 源程序。 汇编语言的编译器为 as。

编译器被成功的移植到不同的处理器平台上。标准 PC LINUX 上的 gcc 是 FOR INTEL CPU 的，而华恒 HHPC852 系列开发套件使用的是 FOR powerpc 系列处理器的 gcc 编译器 powerpc-Linux-gcc 和 powerpc-Linux-as 及其相应的 GNU Binutils 工具集（如 ld 链接工具，objcopy、objdump 等工具）。

gcc 命令的常用选项有：

-ansi	只支持 ANSI 标准的 C 语法。这一选项将禁止 GNU C 的某些特色，例如 asm 或 typeof 关键词。
-c	只编译并生成目标文件。
-DMACRO	以字符串“1”定义 MACRO 宏。
-DMACRO=DEFN	以字符串“DEFN”定义 MACRO 宏。
-E	只运行 C 预编译器。
-g	生成调试信息。GNU 调试器可利用该信息。
-IDIRECTORY	指定额外的头文件搜索路径 DIRECTORY。
-LDIRECTORY	指定额外的函数库搜索路径 DIRECTORY。
-lLIBRARY	连接时搜索指定的函数库 LIBRARY。
-m486	针对 486 进行代码优化。
-o FILE	生成指定的输出文件。用在生成可执行文件时。
-O0	不进行优化处理。
-O 或 -O1	优化生成代码。
-O2	进一步优化。
-O3	比 -O2 更进一步优化，包括 inline 函数。
-shared	生成共享目标文件。通常用在建立共享库时。
-static	禁止使用共享连接。
-UMACRO	取消对 MACRO 宏的定义。
-w	不生成任何警告信息。
-Wall	生成所有警告信息。

ld 文件

编译完成之后，就要执行 ld 进行链接。ld 工具处理 ld 文件。

ld 文件采用 AT&T 链接命令语言写成,用于控制整个链接过程。

GDB

Linux 包含了一个叫 gdb 的 GNU 调试程序. gdb 是一个用来调试 C 和 C++ 程序的强力调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况. Gdb 功能非常强大:

可监视程序中变量的值.

可设置断点以使程序在指定的代码行上停止执行.

支持单步执行等

在命令行上键入 gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话, gdb 将被启动并且你将在屏幕上看到类似的内容:

```
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-Linux".
(gdb)
```

当你启动 gdb 后, 你能在命令行上指定很多的选项. 你也可以以下面的方式来运行 gdb :

```
gdb <fname>
```

当你用这种方式运行 gdb , 你能直接指定想要调试的程序. 这将告诉gdb 装入名为 fname 的可执行文件. 你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件, 或者与一个正在运行的程序相连. 你可以参考 gdb 指南页或在命令行上键入 gdb -h 得到一个有关这些选项的说明的简单列表.

为了使 gdb 正常工作, 你必须使你的程序在编译时包含调试信息. 调试信息包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号. gdb 利用这些信息使源代码和机器码相关联.

在编译时用 -g 选项打开调试选项.

gdb 支持很多的命令使你能实现不同的功能. 这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令, 下面列出了你在用 gdb 调试时会用到的一些命令. 想了解 gdb 的详细使用请参考 gdb 的指南页.

gdb 的常用命令

break NUM 在指定的行上设置断点。
bt 显示所有的调用栈帧。该命令可用来显示函数的调用顺序。
clear 删除设置在特定源文件、特定行上的断点。其用法为：clear FILENAME:NUM。
continue 继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行时。

display EXPR 每次程序停止后显示表达式的值。表达式由程序定义的变量组成。
file FILE 装载指定的可执行文件进行调试。
help NAME 显示指定命令的帮助信息。
info break 显示当前断点清单，包括到达断点处的次数等。
info files 显示被调试文件的详细信息。
info func 显示所有的函数名称。
info local 显示当函数中的局部变量信息。
info prog 显示被调试程序的执行状态。
info var 显示所有的全局和静态变量名称。
kill 终止正被调试的程序。
list 显示源代码段。
make 在不退出 gdb 的情况下运行 make 工具。
next 在不单步执行进入其他函数的情况下，向前执行一行源代码。
print EXPR 显示表达式 EXPR 的值。

gdb 支持很多与 UNIX shell 程序一样的命令编辑特征．你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令．你也能用光标键上下翻动历史命令．

gdb 应用举例

下面用一个实例教你一步步的用 gdb 调试程序．被调试的程序相当的简单，但它展示了 gdb 的典型应用．

下面列出了将被调试的程序．这个程序被称为 greeting ，它显示一个简单的问候，再用反序将它列出．

```
#include  <stdio.h>
main (){
  char my_string[] = "hello there";
  my_print (my_string);
```

```
  my_print2 (my_string);
}
void my_print (char *string){
  printf ("The string is %s\n", string);
}
void my_print2 (char *string){
  char *string2;
  int size, i;
  size = strlen (string);
  string2 = (char *) malloc (size + 1);
  for (i = 0; i < size; i++)
    string2[size - i] = string[i];
  string2[size+1] = `\0';
  printf ("The string printed backward is %s\n", string2);
}
```

用下面的命令编译它:

```
gcc -o test test.c
```

这个程序执行时显示如下结果:

```
The string is hello there
The string printed backward is
```

输出的第一行是正确的, 但第二行打印出的东西并不是我们所期望的. 我们所设想的输出应该是:

```
The string printed backward is ereht olleh
```

由于某些原因, my_print2 函数没有正常工作. 让我们用 gdb 看看问题究竟出在哪儿, 先键入如下命令:

```
gdb greeting
```

【注意】

记得在编译 greeting 程序时把调试选项打开.

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb , 你可以在 gdb 提示符下用 file 命令来载入它:

```
(gdb) file greeting
```

这个命令将载入 greeting 可执行文件就像你在 gdb 命令行里装入它一样. 这时你能用 gdb 的 run 命令来运行 greeting 了. 当它在 gdb 里被运行后结果大约会象这样:

```
(gdb) run
Starting program: /root/greeting
The string is hello there
```

```
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样．问题是，为什么反序打印没有工作？为了找出症结所在，我们可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下键入 list 命令三次，列出源代码：

```
(gdb) list
(gdb) 回车
(gdb) 回车
```

（在 gdb 提示符下按回车健将重复上一个命令。）

要 list 三次是因为一次无法显示全部文件内容，而必须多次才能翻到想要设置断点的文件行处。根据列出的源程序，能看到要设断点的地方在第 24 行，在 gdb 命令行提示符下键入如下命令设置断点：

```
(gdb) break 24
```

gdb 将作出如下的响应：

```
Breakpoint 1 at 0x139: file greeting.c, line 24
(gdb)
```

现在再键入 run 命令，将产生如下的输出：

```
Starting program: /root/greeting
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24
24 string2[size-i]=string[i]
```

通过设置一个观察 string2[size - i] 变量的值的观察点来看出错误是怎样产生的，做法是键入：

```
(gdb) watch string2[size - i]
```

gdb 将作出如下回应：

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步的执行 for 循环了：

```
(gdb) next
```

经过第一次循环后， gdb 告诉我们 string2[size - i] 的值是 `h`．gdb 用如下的显示来告诉你这个信息：

```
Watchpoint 2, string2[size - i]
Old value = 0 `\000'
New value = 104 `h'
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23
23 for (i=0; i<size; i++)
```

这个值正是期望的．后来的数次循环的结果都是正确的．当 i=10 时，表达

式 string2[size - i] 的值等于 `e`, size - i 的值等于 1, 最后一个字符已经拷到新串里了.

如果你再把循环执行下去, 你会看到已经没有值分配给 string2[0] 了, 而它是新串的第一个字符, 因为 malloc 函数在分配内存时把它们初始化为空(null)字符. 所以 string2 的第一个字符是空字符. 这解释了为什么在打印 string2 时没有任何输出了.

现在找出了问题出在哪里, 修正这个错误是很容易的. 你得把代码里写入 string2 的第一个字符的的偏移量改为 size - 1 而不是 size. 这是因为 string2 的大小为 12, 但起始偏移量是 0, 串内的字符从偏移量 0 到 偏移量 10, 偏移量 11 为空字符保留.

附录 D Makefile

在大型的开发项目中，通常有几十到上百个的源文件，如果每次均手工键入 gcc 命令进行编译的话，则会非常不方便。因此，人们通常利用 make 工具来自动完成编译工作。这些工作包括：如果仅修改了某几个源文件，则只重新编译这几个源文件；如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。利用这种自动编译可大大简化开发工作，避免不必要的重新编译。

实际上，make 工具通过一个称为 makefile 的文件来完成并自动维护编译工作。makefile 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。

当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。

makefile 文件是许多编译器，包括 Windows NT 下的编译器维护编译信息的常用方法，只是在集成开发环境中，用户通过友好的界面修改 makefile 文件而已。默认情况下，GNU make 工具在当前工作目录中按如下顺序搜索 makefile：

* GNUmakefile
* makefile
* Makefile

在 UNIX 系统中，习惯使用 Makefile 作为 makfile 文件。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

$ make -f Makefile.debug

Makefile 基本结构

makefile 中一般包含如下内容：

* 需要由 make 工具创建的项目，通常是目标文件和可执行文件。通常使用“目标（target）”一词来表示要创建的项目。
* 要创建的项目依赖于哪些文件。
* 创建每个项目时需要运行的命令。

例如，假设你现在有一个 C++ 源文件 test.C，该源文件包含有自定义的头文件 test.h，则目标文件 test.o 明确依赖于两个源文件：test.c 和 test.h。另外，你可能只希望利用 g++ 命令来生成 test.o 目标文件。

这时，就可以利用如下的 makefile 来定义 test.o 的创建规则：

```
# This makefile just is a example.
# The following lines indicate how test.o depends
# test.C and test.h, and how to create test.o
```

```
test.o: test.c test.h
    g++ -c -g test.C
```

从上面的例子注意到，第一个字符为 # 的行为注释行。第一个非注释行指定 test.o 为目标，并且依赖于 test.c 和 test.h 文件。随后的行指定了如何从目标所依赖的文件建立目标。当 test.c 或 test.h 文件在编译之后又被修改，则 make 工具可自动重新编译 test.o，如果在前后两次编译之间，test.C 和 test.h 均没有被修改，而且 test.o 还存在的话，就没有必要重新编译。这种依赖关系在多源文件的程序编译中尤其重要。通过这种依赖关系的定义，make 工具可避免许多不必要的编译工作。当然，利用 Shell 脚本也可以达到自动编译的效果，但是，Shell 脚本将全部编译任何源文件，包括哪些不必要重新编译的源文件，而 make 工具则可根据目标上一次编译的时间和目标所依赖的源文件的更新时间而自动判断应当编译哪个源文件。

一个 makefile 文件中可定义多个目标，利用 make target 命令可指定要编译的目标，如果不指定目标，
则使用第一个目标。通常，makefile 中定义有 clean 目标，可用来清除编译过程中的中间文件，例如：

```
clean:
    rm -f *.o
```

运行 make clean 时，将执行 rm -f *.o 命令，最终删除所有编译过程中产生的所有中间文件。

Makefile 变量

GNU 的 make 工具除提供有建立目标的基本功能之外，还有许多便于表达依赖性关系以及建立目标的命令的特色。其中之一就是变量或宏的定义能力。如果你要以相同的编译选项同时编译十几个 C 源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义，可避免这种乏味的工作：

```
# Define macros for name of compiler
CC = gcc

# Define a macr o for the CC flags
CCFLAGS = -D_DEBUG -g -m486
# A rule for building a object file
test.o: test.c test.h
    $(CC) -c $(CCFLAGS) test.c
```

在上面的例子中，CC 和 CCFLAGS 就是 make 的变量。GNU make 通常称之为变量，而其他 UNIX 的 make 工具称之为宏，实际是同一个东西。在 makefile 中引用

变量的值时，只需变量名之前添加 $ 符号，如上面的 $(CC) 和 $(CCFLAGS)。

GNU make 的主要预定义变量 i

GNU make 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。表 1-5 给出了一些主要的
预定义变量，除这些变量外，GNU make 还将所有的环境变量作为自己的预定义变量。

$* 不包含扩展名的目标文件名称。
$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。
$< 第一个依赖文件的名称。
$? 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。
$@ 目标的完整名称。
$^ 所有的依赖文件，以空格分开，不包含重复的依赖文件。
$% 如果目标是归档成员，则该变量表示目标的归档成员名称。例如，如果目标名称为 mytarget.so(image.o)，则 $@ 为 mytarget.so，而 $% 为 image.o。
AR 归档维护程序的名称，默认值为 ar。
ARFLAGS 归档维护程序的选项。
AS 汇编程序的名称，默认值为 as。
ASFLAGS 汇编程序的选项。
CC C 编译器的名称，默认值为 cc。
CCFLAGS C 编译器的选项。
CPP C 预编译器的名称，默认值为 $(CC) -E。
CPPFLAGS C 预编译的选项。
CXX C++ 编译器的名称，默认值为 g++。
CXXFLAGS C++ 编译器的选项。
FC FORTRAN 编译器的名称，默认值为 f77。
FFLAGS FORTRAN 编译器的选项。

隐含规则

GNU make 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。

GNU make 支持两种类型的隐含规则：

* 后缀规则（Suffix Rule）。后缀规则是定义隐含规则的老风格方法。后缀规则定义了将一个具有某个后缀的文件（例如，.c 文件）转换为具有另外一种后缀的文件（例如，.o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义，例

如，将 .c 文件转换为 .o 文件的后缀规则可定义为：

```
.c.o:
$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

* 模式规则（pattern rules）。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。模式规则看起来非常类似于正则规则，但在目标名称的前面多了一个 % 号，同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

```
%.c:%.o
$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

Makefile 范例

运行 make

我们知道，直接在 make 命令的后面键入目标名可建立指定的目标，如果直接运行 make，则建立第一个目标。我们还知道可以用 make -f mymakefile 这样的命令指定 make 使用特定的 makefile，而不是默认的 GNUmakefile、makefile 或 Makefile。但 GNU make 命令还有一些其他选项，下面给出了这些选项。

-C DIR 在读取 makefile 之前改变到指定的目录 DIR。
-f FILE 以指定的 FILE 文件作为 makefile。
-h 显示所有的 make 选项。
-i 忽略所有的命令执行错误。
-I DIR 当包含其他 makefile 文件时，可利用该选项指定搜索目录。
-n 只打印要执行的命令，但不执行这些命令。
-p 显示 make 变量数据库和隐含规则。
-s 在执行命令时不显示命令。
-w 在处理 makefile 之前和之后，显示工作目录。
-W FILE 假定文件 FILE 已经被修改。

附录 E 图形界面（GUI）接口函数 API

华恒开发平台提供的 GUI API 仿照 WIN32 API 的接口，使客户能够以最短的时间熟悉并使用它们，实现从 Windows 平台到 Linux 平台开发者的角色转变。其代码目录就是 gui 这个目录。

short initgraph(void)

初始化显示环境,返回结果表示初始化是否成功

void closegraph(void)

关闭显示环境

void clearscreen(void)

清屏

void setpixel(short x,short y,short color)

在（x,y）坐标处画点

short getpixel(short x,short y)

返回（x,y）点的颜色信息

void setmode(CopyMode mode)

设置显示模式

参数 mode 可为下值：

MODE_SRC 从源到目的 COPY

MODE_NOT_SRC 目的为源的反

MODE_SRC_OR_DST 目的为与源相或后的结果

MODE_SRC_AND_DST 目的为与源相与后的结果

MODE_SRC_XOR_DST 目的为与源相异或后的结果

MODE_SRC_OR_NOT_DST 目的为与源的反相或后的结果

MODE_SRC_AND_NOT_DST 目的为目的的反与源相与后的结果

MODE_SRC_XOR_NOT_DST 目的为目的的反与源相异或后的结果

MODE_NOT_SRC_OR_DST 目的为与源的反相或后的结果

MODE_NOT_SRC_AND_DST 目的为与源相与后的结果

MODE_NOT_SRC_XOR_DST 目的为与源的反相异或后的结果

MODE_NOT_SRC_OR_NOT_DST 目的取反与源的反相或后的结果

MODE_NOT_SRC_AND_NOT_DST 目的取反与源的反相与后的结果

MODE_NOT_SRC_XOR_NOT_DST 目的取反与源的反相异或后的结果

CopyMode getmode(void);

获取当前显示模式

void setcolor(short color)

设置显示前景色

UINT getcolor(void)

获取当前显示前景色

void setfillpattern(PatternIndex index)

设置填充模式

PatternIndex getfillpattern(void)

获取当前填充模式

void bar(short x1,short y1,short x2,short y2)

以实填充模式在（x1,y1,x2,y2）绘制矩形

void ellipse(short x1,short y1,short x2,short y2)

在矩形（x1,y1,x2,y2）中绘制椭圆

void line(short x1, short y1, short x2,short y2)

从点（x1,y1）到点（x2,y2）画一条直线

void lineto(short x, short y) 从当前所在点到点（x,y）之间画一条直线

void moveto(short x,short y) 设置当前所在点为点（x,y）

void rectangle(short x1,short y1,short x2,short y2)

在（x1,y1,x2,y2）中绘制矩形

void textout(short x,short y,unsigned char *s)

在（x,y）坐标处输出字符串 s

```
void bitblt(
        short src_x,
        short src_y,
        short w,
        short h,
        short dest_x,
        short dest_y,
        unsigned char *src,
        short src_units_per_line,
        unsigned char *dest,
        short dest_units_per_line
        )
```

位块传送，源地址(src_x,src_y),宽度 w,高度 h，目标地址(dest_x,dest_y)，源块的数据指针 src，src_units_per_line 指明了数据源中每行的宽度，dest 指明了目的数据地址，dest_units_per_line 指明了目的地址方的每行的宽度

void ShowBMP(char *filename,short x,short y)

在 x，y 处显示位图

void draw_bmp(short sx, short sy, short rwidth, short height, char* pixel)

位图显示函数在指定的 sx,sy 处显示每行逻辑宽度为 rwidth 字节的位图，pixel 中指定了位图的图象信息，通常的使用可以参考 ShowBMP 中的使用

void V_scroll_screen(short height)

竖直方向滚动屏幕，以 height 为单位。height>0，屏幕上滚；height<0，屏幕下滚。

void H_scroll_screen(short height)

水平方向滚动屏幕，以 height 为单位。height>0，屏幕向左滚动；height<0，屏幕向右滚动。

附录 F 启动代码分析

这部分的分析请结合正文的“内核编译”一节阅读。

一、最先的启动代码来自：

/HHARM2440/kernel/arch/arm/boot/compressed/head.S

对应的 ld 文件为：

/HHARM2440/kernel/arch/arm/boot/compressed/vmlinux.lds

```
vim vmlinux.lds
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x30008000;        /*这是 head.S 的执行地址*/
    _load_addr = .;

    . = 0;
    _text = .;

    .text : {
        _start = .;
        *(.start)          【.start 段的内容放在这里】
        ……
```

下面摘录 head.S 的主体代码列出：

```
                .section ".start", #alloc, #execinstr
/* * sort out different calling conventions */
                .align
start:
                .type    start,#function
                .rept    8
                mov      r0, r0
                .endr

                b        1f
                .word    0x016f2818              @ Magic numbers to help the loader
```

```
                    .word      start                          @  absolute  load/run  zImage
address
                    .word      _edata                          @ zImage end address
1:                  mov        r7, r1                         @ save architecture ID
                    mov        r8, #0                          @ save r0

                    teqp       pc, #0x0c000003                @ turn off interrupts

                    /*
                     * Note that some cache flushing and other stuff may
                     * be needed here - is there an Angel SWI call for this?
                     */

                    /*
                     * some architecture specific code can be inserted
                     * by the linker here, but it should preserve r7 and r8.
                     */

                    .text
                    adr        r0, LC0
                    ldmia      r0, {r1, r2, r3, r4, r5, r6, ip, sp}
                    subs       r0, r0, r1                    @ calculate the delta offset

                    teq        r0, #0                         @ if delta is zero, we're
                    beq        not_relocated                 @ running at the address we
                                                                @ were linked at.

                    /*
                     * We're running at a different address.    We need to fix
                     * up various pointers:
                     *      r5 - zImage base address
                     *      r6 - GOT start
                     *      ip - GOT end
                     */
                    add        r5, r5, r0
                    add        r6, r6, r0
                    add        ip, ip, r0
```

```
                /*
                 * If we're running fully PIC === CONFIG_ZBOOT_ROM = n,
                 * we need to fix up pointers into the BSS region.
                 *      r2 - BSS start
                 *      r3 - BSS end
                 *      sp - stack pointer
                 */
                add     r2, r2, r0
                add     r3, r3, r0
                add     sp, sp, r0

                /*
                 * Relocate all entries in the GOT table.
                 */
1:              ldr     r1, [r6, #0]
                add     r1, r1, r0
                str     r1, [r6], #4
                cmp     r6, ip
                blo     1b
not_relocated:  mov     r0, #0
1:              str     r0, [r2], #4            @ clear bss
                str     r0, [r2], #4
                str     r0, [r2], #4
                str     r0, [r2], #4
                cmp     r2, r3
                blo     1b

                /*
                 * The C runtime environment should now be setup
                 * sufficiently.   Turn the cache on, set up some
                 * pointers, and start decompressing.
                 */
                bl      cache_on

                mov     r1, sp                  @ malloc space above stack
                add     r2, sp, #0x10000        @ 64k max
```

```
/*
 * Check to see if we will overwrite ourselves.
 *      r4 = final kernel address
 *      r5 = start of this image
 *      r2 = end of malloc space (and therefore this image)
 * We basically want:
 *      r4 >= r2 -> OK
 *      r4 + image length <= r5 -> OK
 */
                cmp     r4, r2
                bhs     wont_overwrite
                add     r0, r4, #4096*1024      @ 4MB largest kernel size
                cmp     r0, r5
                bls     wont_overwrite
                mov     r5, r2                  @ decompress after malloc
space
                mov     r0, r5
                mov     r3, r7
                bl      decompress_kernel

                add     r0, r0, #127
                bic     r0, r0, #127            @ align the kernel length
/*
 * r0       = decompressed kernel length
 * r1-r3    = unused
 * r4       = kernel execution address
 * r5       = decompressed kernel start
 * r6       = processor ID
 * r7       = architecture ID
 * r8-r14 = unused
 */
                add     r1, r5, r0              @ end of decompressed kernel
                adr     r2, reloc_start
                ldr     r3, LC1
                add     r3, r2, r3
1:              ldmia   r2!, {r8 - r13}         @ copy relocation code
```

```
                    stmia     r1!, {r8 - r13}
                    ldmia     r2!, {r8 - r13}
                    stmia     r1!, {r8 - r13}
                    cmp       r2, r3
                    blo       1b

                    bl        cache_clean_flush
                    add       pc, r5, r0                @ call relocation code
/*
 * We're not in danger of overwriting ourselves.   Do this the simple way.
 *
 * r4       = kernel execution address
 * r7       = architecture ID
 */
wont_overwrite:     mov       r0, r4
                    mov       r3, r7
                    bl        decompress_kernel
                    b         call_kernel

                    .type     LC0, #object
LC0:                .word     LC0                       @ r1
                    .word     __bss_start               @ r2
                    .word     _end                      @ r3
                    .word     _load_addr                @ r4
                    .word     _start                    @ r5
                    .word     _got_start                @ r6
                    .word     _got_end                  @ ip
                    .word     user_stack+4096           @ sp
LC1:                .word     reloc_end - reloc_start
                    .size     LC0, . - LC0
/*
 * Turn on the cache.   We need to setup some page tables so that we
 * can have both the I and D caches on.
 *
 * We place the page tables 16k down from the kernel execution address,
 * and we hope that nothing else is using it.   If we're using it, we
 * will go pop!
```

```
 *
 * On entry,
 *    r4 = kernel execution address
 *    r6 = processor ID
 *    r7 = architecture number
 *    r8 = run-time address of "start"
 * On exit,
 *    r1, r2, r3, r8, r9, r12 corrupted
 * This routine must preserve:
 *    r4, r5, r6, r7
 */
                .align  5
cache_on:       mov     r3, #8                  @ cache_on function
                b       call_cache_fn

__setup_mmu:

                sub     r3, r4, #16384          @ Page directory size
                bic     r3, r3, #0xff           @ Align the pointer
                bic     r3, r3, #0x3f00
/*
 * Initialise the page tables, turning on the cacheable and bufferable
 * bits for the RAM area only.
 */
                mov     r0, r3
                mov     r8, r0, lsr #18
                mov     r8, r8, lsl #18         @ start of RAM
                add     r9, r8, #0x10000000     @ a reasonable RAM size
                mov     r1, #0x12
                orr     r1, r1, #3 << 10
                add     r2, r3, #16384
1:              cmp     r1, r8                  @ if virt > start of RAM
#ifdef CONFIG_XSCALE_CACHE_ERRATA
                orrhs   r1, r1, #0x08           @ set cacheable, not bufferable
#else
                orrhs   r1, r1, #0x0c           @ set cacheable, bufferable
#endif
```

```
                cmp     r1, r9                  @ if virt > end of RAM
                bichs   r1, r1, #0x0c           @ clear cacheable, bufferable
                str     r1, [r0], #4            @ 1:1 mapping
                add     r1, r1, #1048576
                teq     r0, r2
                bne     1b
/*
 * If ever we are running from Flash, then we surely want the cache
 * to be enabled also for our execution instance...   We map 2MB of it
 * so there is no map overlap problem for up to 1 MB compressed kernel.
 * If the execution is in RAM then we would only be duplicating the above. */
                mov     r1, #0x1e
                orr     r1, r1, #3 << 10
                mov     r2, pc, lsr #20
                orr     r1, r1, r2, lsl #20
                add     r0, r3, r2, lsl #2
                str     r1, [r0], #4
                add     r1, r1, #1048576
                str     r1, [r0]
                mov     pc, lr

__armv4_cache_on:
                mov     r12, lr
                bl      __setup_mmu
                mov     r0, #0
                mcr     p15, 0, r0, c7, c10, 4  @ drain write buffer
                mcr     p15, 0, r0, c8, c7      @ flush I,D TLBs
                mcr     p15, 0, r3, c2, c0      @ load page table pointer
                mov     r0, #-1
                mcr     p15, 0, r0, c3, c0      @ load domain access register
                mrc     p15, 0, r0, c1, c0
                orr     r0, r0, #0x1000         @ I-cache enable
#ifndef DEBUG
                orr     r0, r0, #0x003d         @ Write buffer, mmu
#endif
                mcr     p15, 0, r0, c1, c0
                mov     pc, r12
```

```
__arm6_cache_on:
                mov     r12, lr
                bl      __setup_mmu
                mov     r0, #0
                mcr     p15, 0, r0, c7, c0, 0   @ invalidate whole cache v3
                mcr     p15, 0, r0, c5, c0, 0   @ invalidate whole TLB v3
                mcr     p15, 0, r3, c2, c0      @ load page table pointer
                mov     r0, #-1
                mcr     p15, 0, r0, c3, c0
                mov     r0, #0x3d
                mcr     p15, 0, r0, c1, c0, 0
                mov     pc, r12

/*
 * All code following this line is relocatable.  It is relocated by
 * the above code to the end of the decompressed kernel image and
 * executed there.  During this time, we have no stacks.
 *
 * r0      = 压缩内核的文件长度
 * r1-r3   = unused
 * r4      = 内核解压后的起始地址=0xC0008000
 * r5      = decompressed kernel start
 * r6      = processor ID
 * r7      = architecture ID
 * r8-r14 = unused
 */
                .align  5
reloc_start:    add     r8, r5, r0
                debug_reloc_start
                mov     r1, r4
1:
                .rept   4
                ldmia   r5!, {r0, r2, r3, r9 - r13}     @ relocate kernel
                stmia   r1!, {r0, r2, r3, r9 - r13}
                .endr

                cmp     r5, r8
```

```
                blo         1b
                debug_reloc_end

call_kernel:    bl          cache_clean_flush
                bl          cache_off
                mov         r0, #0
                mov         r1, r7                      @ restore architecture
number
                mov         pc, r4                      @ call kernel
```

二、真正内核的启动代码来自

/HHARM2440/linux-2.4.20/arch/arm/kernel/head-armv.S

对应的 ld 文件为：

/HHARM2440/linux-2.4.20/arch/arm/vmlinux.lds

下面看看 head-armv.S 文件的内容：

```
/*
 *   linux/arch/arm/kernel/head-armv.S
 *
 *   Copyright (C) 1994-1999 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 *   32-bit kernel startup code for all architectures
 */
#include <linux/config.h>
#include <linux/linkage.h>

#include <asm/assembler.h>
#include <asm/mach-types.h>
#include <asm/mach/arch.h>

#define K(a,b,c)     ((a) << 24 | (b) << 12 | (c))

/*
 * We place the page tables 16K below TEXTADDR.   Therefore, we must make sure
```

```
 * that TEXTADDR is correctly set.   Currently, we expect the least significant
 * "short" to be 0x8000, but we could probably relax this restriction to
 * TEXTADDR > PAGE_OFFSET + 0x4000
 *
 * Note that swapper_pg_dir is the virtual address of the page tables, and
 * pgtbl gives us a position-independent reference to these tables.   We can
 * do this because stext == TEXTADDR
 *
 * swapper_pg_dir, pgtbl and krnladr are all closely related.
 */
#if (TEXTADDR & 0xffff) != 0x8000
#error TEXTADDR must start at 0xXXXX8000
#endif

		.globl	SYMBOL_NAME(swapper_pg_dir)
		.equ SYMBOL_NAME(swapper_pg_dir), TEXTADDR - 0x4000

		.macro	pgtbl, reg, rambase
		adr	\reg, stext
		sub	\reg, \reg, #0x4000
		.endm

/*
 * Since the page table is closely related to the kernel start address, we
 * can convert the page table base address to the base address of the section
 * containing both.
 */
		.macro	krnladr, rd, pgtable, rambase
		bic	\rd, \pgtable, #0x000ff000
		.endm

/*
 *  Kernel startup entry point.
 *
 * The rules are:
 *  r0		- should be 0
 *  r1		- unique architecture number
```

```
 *   MMU        - off
 *   I-cache - on or off
 *   D-cache - off
 *
 * See linux/arch/arm/tools/mach-types for the complete list of numbers
 * for r1.
 */
            .section ".text.init",#alloc,#execinstr
            .typestext, #function
ENTRY(stext)
            mov r12, r0
/*
 * NOTE!   Any code which is placed here should be done for one of
 * the following reasons:
 *
 *   1. Compatability with old production boot firmware (ie, users
 *       actually have and are booting the kernel with the old firmware)
 *       and therefore will be eventually removed.
 *   2. Cover the case when there is no boot firmware.   This is not
 *       ideal, but in this case, it should ONLY set r0 and r1 to the
 *       appropriate value.
 */
#if defined(CONFIG_ARCH_NETWINDER)
/*
 * Compatability cruft for old NetWinder NeTTroms.   This
 * code is currently scheduled for destruction in 2.5.xx
 */
            .rept 8
            mov r0, r0
            .endr

            adr  r2, 1f
            ldmdb     r2, {r7, r8}
            and  r3, r2, #0xc000
            teq  r3, #0x8000
            beq  __entry
            bic  r3, r2, #0xc000
```

```
        orr   r3, r3, #0x8000
        mov r0, r3
        mov r4, #64
        sub   r5, r8, r7
        b     1f

        .word     _stext
        .word     __bss_start

1:
        .rept 4
        ldmia     r2!, {r6, r7, r8, r9}
        stmia     r3!, {r6, r7, r8, r9}
        .endr
        subs r4, r4, #64
        bcs   1b
        movs      r4, r5
        mov r5, #0
        movne     pc, r0

        mov r1, #MACH_TYPE_NETWINDER      @ (will go in 2.5)
        mov r12, #2 << 24                 @ scheduled for removal in 2.5.xx
        orr   r12, r12, #5 << 12
__entry:
#endif
#if defined(CONFIG_ARCH_L7200)
/*
 * FIXME - No bootloader, so manually set 'r1' with our architecture number.
 */
        mov r1, #MACH_TYPE_L7200
#endif

        mov r0, #F_BIT | I_BIT | MODE_SVC @ make sure svc mode
        msr  cpsr_c, r0             @ and all irqs disabled
        bl    __lookup_processor_type
        teq   r10, #0                     @ invalid processor?
        moveq     r0, #'p'                @ yes, error 'p'
```

```
            beq   __error
            bl    __lookup_architecture_type
            teq   r7, #0                    @ invalid architecture?
            moveq     r0, #'a'              @ yes, error 'a'
            beq   __error
            bl    __create_page_tables
            adr   lr, __ret             @ return address
            add   pc, r10, #12              @ initialise processor
                                        @ (return control reg)

            .type__switch_data, %object
__switch_data:      .long       __mmap_switched
            .long       SYMBOL_NAME(__bss_start)
            .long       SYMBOL_NAME(_end)
            .long       SYMBOL_NAME(processor_id)
            .long       SYMBOL_NAME(__machine_arch_type)
            .long       SYMBOL_NAME(cr_alignment)
            .long       SYMBOL_NAME(init_task_union)+8192

            .type__ret, %function
__ret:          ldr   lr, __switch_data
            mcr  p15, 0, r0, c1, c0
            mov r0, r0
            mov r0, r0
            mov r0, r0
            mov pc, lr

            /*
             * This code follows on after the page
             * table switch and jump above.
             *
             * r0    = processor control register
             * r1    = machine ID
             * r9    = processor ID
             */
            .align      5
__mmap_switched:
```

```
        adr   r3, __switch_data + 4
        ldmia     r3, {r4, r5, r6, r7, r8, sp}@ r2 = compat
                                  @ sp = stack pointer

        mov fp, #0                    @ Clear BSS (and zero fp)
1:      cmp r4, r5
        strcc fp, [r4],#4
        bcc   1b

        str   r9, [r6]            @ Save processor ID
        str   r1, [r7]            @ Save machine type
#ifdef CONFIG_ALIGNMENT_TRAP
        orr   r0, r0, #2          @ ...........A.
#endif
        bic   r2, r0, #2          @ Clear 'A' bit
        stmia     r8, {r0, r2}            @ Save control register values
        b     SYMBOL_NAME(start_kernel)



/*
 * Setup the initial page tables.   We only setup the barest
 * amount which are required to get the kernel running, which
 * generally means mapping in the kernel code.
 *
 * We only map in 4MB of RAM, which should be sufficient in
 * all cases.
 *
 * r5 = physical address of start of RAM
 * r6 = physical IO address
 * r7 = byte offset into page tables for IO
 * r8 = page table flags
 */
__create_page_tables:
        pgtbl     r4, r5                  @ page table address

        /*
```

```
  * Clear the 16K level 1 swapper page table
  */
mov r0, r4
mov r3, #0
add  r2, r0, #0x4000
1:        str    r3, [r0], #4
str    r3, [r0], #4
str    r3, [r0], #4
str    r3, [r0], #4
teq   r0, r2
bne   1b

/*
  * Create identity mapping for first MB of kernel to
  * cater for the MMU enable.    This identity mapping
  * will be removed by paging_init()
  */
krnladr    r2, r4, r5              @ start of kernel
add  r3, r8, r2              @ flags + kernel base
str    r3, [r4, r2, lsr #18]         @ identity mapping

/*
  * Now setup the pagetables for our kernel direct
  * mapped region.    We round TEXTADDR down to the
  * nearest megabyte boundary.
  */
add  r0, r4, #(TEXTADDR & 0xff000000) >> 18 @ start of kernel
bic   r2, r3, #0x00f00000
str    r2, [r0]                  @ PAGE_OFFSET + 0MB
add  r0, r0, #(TEXTADDR & 0x00f00000) >> 18
str    r3, [r0], #4                  @ KERNEL + 0MB
add  r3, r3, #1 << 20
str    r3, [r0], #4                  @ KERNEL + 1MB
add  r3, r3, #1 << 20
str    r3, [r0], #4                  @ KERNEL + 2MB
add  r3, r3, #1 << 20
str    r3, [r0], #4                  @ KERNEL + 3MB
```

```
            /*
             * Ensure that the first section of RAM is present.
             * we assume that:
             *   1. the RAM is aligned to a 32MB boundary
             *   2. the kernel is executing in the same 32MB chunk
             *        as the start of RAM.
             */
            bic   r0, r0, #0x01f00000 >> 18     @ round down
            and   r2, r5, #0xfe000000       @ round down
            add   r3, r8, r2              @ flags + rambase
            str   r3, [r0]

            bic   r8, r8, #0x0c                 @ turn off cacheable
                                          @ and bufferable bits
#ifdef CONFIG_DEBUG_LL
            /*
             * Map in IO space for serial debugging.
             * This allows debug messages to be output
             * via a serial console before paging_init.
             */
            add   r0, r4, r7
            rsb   r3, r7, #0x4000         @ PTRS_PER_PGD*sizeof(long)
            cmp r3, #0x0800
            addge       r2, r0, #0x0800
            addltr2, r0, r3
            orr   r3, r6, r8
1:          str   r3, [r0], #4
            add   r3, r3, #1 << 20
            teq   r0, r2
            bne   1b
#if defined(CONFIG_ARCH_NETWINDER) || defined(CONFIG_ARCH_CATS)
            /*
             * If we're using the NetWinder, we need to map in
             * the 16550-type serial port for the debug messages
             */
            teq   r1, #MACH_TYPE_NETWINDER
```

```
		teqne	r1, #MACH_TYPE_CATS
		bne	1f
		add	r0, r4, #0x3fc0
		mov	r3, #0x7c000000
		orr	r3, r3, r8
		str	r3, [r0], #4
		add	r3, r3, #1 << 20
		str	r3, [r0], #4
1:
#endif
#endif
#ifdef CONFIG_ARCH_RPC
		/*
		 * Map in screen at 0x02000000 & SCREEN2_BASE
		 * Similar reasons here - for debug.    This is
		 * only for Acorn RiscPC architectures.
		 */
		add	r0, r4, #0x80			@ 02000000
		mov	r3, #0x02000000
		orr	r3, r3, r8
		str	r3, [r0]
		add	r0, r4, #0x3600			@ d8000000
		str	r3, [r0]
#endif
		mov	pc, lr


/*
 * Exception handling.    Something went wrong and we can't
 * proceed.    We ought to tell the user, but since we
 * don't have any guarantee that we're even running on
 * the right architecture, we do virtually nothing.
 * r0 = ascii error character:
 *    a = invalid architecture
 *    p = invalid processor
 *    i = invalid calling convention
 *
```

```
 * Generally, only serious errors cause this.
 */
__error:
#ifdef CONFIG_DEBUG_LL
		mov r8, r0				@ preserve r0
		adr	r0, err_str
		bl	printascii
		mov r0, r8
		bl	printch
#endif
#ifdef CONFIG_ARCH_RPC
/*
 * Turn the screen red on a error - RiscPC only.
 */
		mov r0, #0x02000000
		mov r3, #0x11
		orr	r3, r3, r3, lsl #8
		orr	r3, r3, r3, lsl #16
		str	r3, [r0], #4
		str	r3, [r0], #4
		str	r3, [r0], #4
		str	r3, [r0], #4
#endif
1:		mov r0, r0
		b	1b

#ifdef CONFIG_DEBUG_LL
err_str:	.asciz	"\nError: "
		.align
#endif

/*
 * Read processor ID register (CP#15, CR0), and look up in the linker-built
 * supported processor list.   Note that we can't use the absolute addresses
 * for the __proc_info lists since we aren't running with the MMU on
 * (and therefore, we are not in the correct address space).   We have to
 * calculate the offset.
```

```
 *
 * Returns:
 *    r5, r6, r7 corrupted
 *    r8    = page table flags
 *    r9    = processor ID
 *    r10 = pointer to processor structure
 */
__lookup_processor_type:
        adr   r5, 2f
        ldmia     r5, {r7, r9, r10}
        sub   r5, r5, r10             @ convert addresses
        add   r7, r7, r5              @ to our address space
        add   r10, r9, r5
        mrc   p15, 0, r9, c0, c0          @ get processor id
1:      ldmia     r10, {r5, r6, r8}           @ value, mask, mmuflags
        and   r6, r6, r9              @ mask wanted bits
        teq   r5, r6
        moveq     pc, lr
        add   r10, r10, #36               @ sizeof(proc_info_list)
        cmp r10, r7
        blt   1b
        mov r10, #0                       @ unknown processor
        mov pc, lr

/*
 * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
 * more information about the __proc_info and __arch_info structures.
 */
2:      .long     __proc_info_end
        .long     __proc_info_begin
        .long     2b
        .long     __arch_info_begin
        .long     __arch_info_end

/*
 * Lookup machine architecture in the linker-build list of architectures.
 * Note that we can't use the absolute addresses for the __arch_info
```

```
 * lists since we aren't running with the MMU on (and therefore, we are
 * not in the correct address space).  We have to calculate the offset.
 *
 *   r1 = machine architecture number
 * Returns:
 *   r2, r3, r4 corrupted
 *   r5 = physical start address of RAM
 *   r6 = physical address of IO
 *   r7 = byte offset into page tables for IO
 */
__lookup_architecture_type:
		adr	r4, 2b
		ldmia	r4, {r2, r3, r5, r6, r7}	@ throw away r2, r3
		sub	r5, r4, r5		@ convert addresses
		add	r4, r6, r5		@ to our address space
		add	r7, r7, r5
1:		ldr	r5, [r4]		@ get machine type
		teq	r5, r1
		beq	2f
		add	r4, r4, #SIZEOF_MACHINE_DESC
		cmp	r4, r7
		blt	1b
		mov	r7, #0			@ unknown architecture
		mov	pc, lr
2:		ldmib	r4, {r5, r6, r7}	@ found, get results
		mov	pc, lr
```

附录 G 参考资料

A) S3C2440X 32-bit MICROPROCESSOR User’s Manual；
B) 《UNIX 环境高级编程》，W.Richard Stevens 著，机械工业出版社；
C) 《Linux 设备驱动程序》，ALESSANDRO RUBINI 著，LISOLEG 译，中国电力出版社；
D) 《嵌入式 LINUX 设计与应用》（清华大学出版社 2002 年出版）。

售后服务与技术支持

在使用华恒嵌入式 Linux 开发套件进行开发时所遇到的一切问题，均可在华恒嵌入式 Linux 技术论坛：http://bbs.hhcn.com/ 站点上得到专家们快捷、准确的回答。

另外，还可以发 mail 到华恒公司工程师组电子信箱 hharm-support@hhcn.com，或者直接致电华恒公司，您都会得到及时的答复。

http://www.hhcn.org 上还提供支持华恒嵌入式 Linux 开发板应用开发的技术资料、源代码、开发源代码的 GNU 项目以及开发套件软件的升级版本和补丁，使得基于华恒嵌入式 Linux 开发板进行产品开发更快捷、便利。还有，技术支持论坛上提供我们公司各种系列开发平台的“常见问题解答”的链接。

★提示★ 希望大家首先看看我们网站上的“常见问题解答”，也许您以后遇到同样的情况，问题立即就会得到解答。

http://www.hhcn.com/chinese/hharmfaq.html ：华恒 ARM 产品 FAQ
http://www.hhcn.com/chinese/hhco5272r1faq.html ：华恒 Coldfire 产品 FAQ
http://www.hhcn.com/chinese/hhppcfaq.html ：华恒 Power PC 产品 FAQ
http://www.hhcn.com/chinese/hhdrez328_faq.html ：华恒 **DragonBall** 产品 FAQ
http://www.hhcn.com/chinese/embedlinux-res.html ：嵌入式 Linux 开发资源

华恒嵌入式 Linux 开发套件,整体保修期为六个月，第一个月可免费更换，但 CPU 烧坏、flash 烧毁、SDRAM、LCD 屏,电源、插座、JTAG 卡及电缆等易耗件及人为损坏不在保修范围之内。

如果您在使用华恒产品时遇到故障，请首先与我们的售后技术支持联系，需要返回华恒公司维修的，请认真填写下面表格中您的联系方式(或夹上您的名片)、返修清单和故障现像，把此表格打印出来放到包裹里。因为您寄过来的包裹到华恒公司时，写在包裹外面的您的地址可能已经看不清楚了，或您的地址已经变更，谢谢合作。

华恒科技的联系方式请见本手册的最后一页，并与售后技术支持人员联系确认“收信人”。

我们会在收到返修器件的三个工作日内给出初步结果，特殊情况(像 CPU 等重要烧坏)一周时间给出初步结果。

警告：

请务必注意静电的防护。超过任何最大承受值，均会对产品产生永久损害。同时，不推荐在临界状态使用产品。

客 户 返 修 单

<table>
<tr><td>单位名称</td><td colspan="3"></td></tr>
<tr><td>单位地址</td><td colspan="3"></td></tr>
<tr><td>联 系 人</td><td></td><td>联系电话</td><td></td></tr>
<tr><td>邮　　编</td><td></td><td>E-mail</td><td></td></tr>
<tr><td>返修事由、故障现像</td><td colspan="3"></td></tr>
<tr><td>返修器件清单</td><td colspan="3">1.
2.
3.
4.
5.</td></tr>
<tr><td>备注</td><td colspan="3"></td></tr>
</table>

公司简介：

华恒科技是获得双软企业认证及高新技术企业认证的嵌入式 Linux 开发平台及 OEM 板级硬件提供商，是中国软件协会嵌入式分会理事单位、安徽省软件协会会员、安徽省火炬计划、产业化项目支持企业。公司成立于 1998 年，同步于全球嵌入式 Linux 发展的萌芽期，是国内最早进行嵌入式 Linux 软硬件研究与开发的企业。

华恒科技是摩托罗拉半导体（现 freescale）的全球设计联盟成员、ADI 的 DSP 技术合作伙伴以及 ARM 公司 ARM Connected Community 成员伙伴，与这些处理器巨头进行深度技术研发合作；华恒科技与其他国际知名的半导体公司如 Intel、三星、威盛、华邦等也保持商务与技术方面的联系与合作；目前，华恒科技拥有基于 68K/Coldfire、PowerPC、ARM、ADSP 四大主流嵌入式处理器系列的开发平台，实现了跨越通信设备、消费电子、工业控制三大应用领域的技术，是国内唯一一家能够实现如此丰富产品的嵌入式 Linux 软硬件专业提供商。

公司主页：

http://www.hhcn.com　　华恒科技产品网站
http://www.hhcn.org　　华恒科技技术资料网站

公司论坛：

http://bbs.hhcn.com

技术支持邮箱：

hharm-support@hhcn.com　　ARM 系列产品技术支持邮箱
hhbf-support@hhcn.com　　Blackfin 系列产品技术支持邮箱
hhcf-support@hhcn.com　　Coldfire 系列产品技术支持邮箱
hhppc-support@hhcn.com　　Power PC 系列产品技术支持邮箱

总部地址：安徽省合肥市黄山路 626 号国家高新技术产业开发区银河大厦 C 座 4 层　华恒科技
邮　　编：230088
总部电话：0551-5325652/5325653/5325631/5325173/5333155/5333156/ 5333157
总部传真：0551-5325323
电子邮件：market@hhcn.com